Gradle

官方文档 中文翻译

VISIT…

目錄

关于本书 1.1
介绍 1.2
    关于这本指南 1.2.1
概述 1.3
    特点 1.3.1
    为什么用 Groovy? 1.3.2
教程 1.4
安装Gradle 1.5
    准备阶段 1.5.1
    下载与安装 1.5.2
    JVM选项 1.5.3
排除故障 1.6
构建脚本基础 1.7
    Projects 和 tasks 1.7.1
    Hello world 1.7.2
    快捷的任务定义 1.7.3
    构建脚本代码 1.7.4
    任务依赖 1.7.5
    动态任务 1.7.6
    使用已经存在的任务 1.7.7
    短标记法 1.7.8
    自定义任务属性 1.7.9
    调用 Ant 任务 1.7.10
    使用方法 1.7.11
    默认任务 1.7.12
    通过 DAG 配置 1.7.13
Java 构建入门 1.8
    Java 插件 1.8.1
    一个基础的 Java 项目 1.8.2
        建立项目 1.8.2.1

外部的依赖 1.8.2.2
定制项目 1.8.2.3
发布 JAR 文件 1.8.2.4
创建 Eclipse 项目 1.8.2.5
总结 1.8.2.6
多项目的 Java 构建 1.8.3
定义一个多项目构建 1.8.3.1
通用配置 1.8.3.2
项目之间的依赖 1.8.3.3
创建一个发行版本 1.8.3.4
依赖管理的基础知识 1.9
什么是依赖管理 1.9.1
声明你的依赖 1.9.2
依赖配置 1.9.3
外部的依赖 1.9.4
仓库 1.9.5
发布 artifacts 1.9.6
下一步? 1.9.7
Groovy 快速入门 1.10
一个基本的 Groovy 项目 1.10.1
总结 1.10.2
网页应用快速入门 1.11
构建一个 WAR 文件 1.11.1
运行 Web 应用 1.11.2
总结 1.11.3
使用 Gradle 命令行 1.12
多任务调用 1.12.1
排除任务 1.12.2
失败后继续执行构建 1.12.3
简化任务名 1.12.4
选择文件构建 1.12.5
获取构建信息 1.12.6
项目列表 1.12.6.1
任务列表 1.12.6.2

获取任务具体信息 1.12.6.3
获取依赖列表 1.12.6.4
查看特定依赖 1.12.6.5
获取项目属性列表 1.12.6.6
构建日志 1.12.6.7
使用 Gradle 图形界面 1.13
任务树 1.13.1
收藏夹 1.13.2
命令行 1.13.3
设置 1.13.4
编写构建脚本 1.14
Gradle 构建语言 1.14.1
项目 API 1.14.2
标准项目属性 1.14.2.1
脚本 API 1.14.3
声明变量 1.14.4
局部变量 1.14.4.1
扩展属性 1.14.4.2
Groovy 基础 1.14.5
Groovy JDK 1.14.5.1
属性存取器 1.14.5.2
可有可无的圆括号 1.14.5.3
List 和 Map 集合 1.14.5.4
闭合作为方法的最后一个参数 1.14.5.5
闭合委托对象 1.14.5.6
深入了解 Tasks 1.15
定义 tasks 1.15.1
定位 tasks 1.15.2
配置 tasks 1.15.3
给 task 加入依赖 1.15.4
给 tasks 排序 1.15.5
给 task 加入描述 1.15.6
替换 tasks 1.15.7

跳过 tasks 1.15.8
跳过 up-to-date 的任务 1.15.9
Task 规则 1.15.10
终止 tasks 1.15.11
补充 1.15.12
Gradle 属性 和 system 属性 1.15.12.1
使用其他的脚本配置项目 1.15.12.2
使用其他的脚本配置任意对象 1.15.12.3
配置任意对象 1.15.12.4
缓存 1.15.12.5
文件操作 1.16
定位文件 1.16.1
文件集合 1.16.2
文件树 1.16.3
使用一个归档文件的内容作为文件树 1.16.4
指定一组输入文件 1.16.5
复制文件 1.16.6
使用同步任务 1.16.7
创建归档文件 1.16.8
使用 Ant 插件 1.17
使用 Ant 任务和 Ant 类型的构建 1.17.1
在构建中使用自定义 Ant 任务 1.17.1.1
导入一个 Ant 构建 1.17.2
Ant 的属性与引用 1.17.3
API 1.17.4
Logging 1.18
选择日志等级 1.18.1
编写自己的日志信息 1.18.2
外部工具和库的log 1.18.3
改变 Gradle 记录的内容 1.18.4
Gradle的守护进程 1.19
什么是Gradle的守护进程 1.19.1
管理和配置 1.19.2
如何启用的摇篮守护进程 1.19.2.1

如何禁用Gradle的守护进程 1.19.2.2
怎样抑制“please consider using the Gradle Daemon”消息 1.19.2.3
为什么会在机器上出现不只一个守护进程 1.19.2.4
守护进程占用多大内存并且能不能给它更大的内存? 1.19.2.5
如何停止守护进程 1.19.2.6
守护进程何时会出错 1.19.2.7
什么时候使用Gradle守护进程 1.19.3
工具和集成开发环境 1.19.4
摇篮守护进程如何使构建速度更快 1.19.5
未来可能的改进 1.19.5.1
Gradle Plugins 1.20
插件的作用 1.20.1
插件的类型 1.20.2
应用插件 1.20.3
脚本插件 1.20.3.1
二进制插件 1.20.3.2
二进制插件的位置 1.20.3.2.1
使用构建脚本块应用插件 1.20.4
使用插件的插件DSL 1.20.5
插件DSL的限制 1.20.5.1
约束语法 1.20.5.2
只能在构建脚本中使用 1.20.5.3
不能与subjects{},allprojects{}等结合使用 1.20.5.4
查找社区插件 1.20.6
更多关于插件 1.20.7
Gradle插件规范 1.21
语言插件 1.21.1
孵化中的语言插件 1.21.2
集成插件 1.21.3
孵化中的集成插件 1.21.4
软件开发插件 1.21.5
孵化中的软件开发插件 1.21.6
基础插件 1.21.7

第三方插件 1.21.8
Java 插件 1.22
使用 1.22.1
资源集 1.22.2
任务 1.22.3
项目布局 1.22.4
依赖管理 1.22.5
公共配置 1.22.6
使用资源集工作 1.22.7
资源集属性 1.22.7.1
定义新的资源集 1.22.7.2
资源集例子 1.22.7.3
Javadoc 1.22.8
清除 1.22.9
资源 1.22.10
编译 Java 1.22.11
增量 Java 编译 1.22.12
测试 1.22.13
测试执行 1.22.13.1
测试调试 1.22.13.2
测试过滤 1.22.13.3
通过系统属性执行单独测试 1.22.13.4
测试检测 1.22.13.5
测试分组 1.22.13.6
测试报告 1.22.13.7
TestNG 的参数化方法和报告 1.22.13.7.1
公共值 1.22.13.8
Jar 1.22.14
Manifest 1.22.14.1
上传 1.22.15
War插件 1.23
使用 1.23.1
任务 1.23.2
项目布局 1.23.3

依赖管理	1.23.4
公共配置	1.23.5
War	1.23.6
定制War	1.23.7
Ear插件	1.24

Gradle User Guide 中文版

阅读地址

- Gradle User Guide 中文版 目前正在更新到官方**4.6**版本 (branch/#22), 将会在翻译完主要内容后合并的该分支. 因工作量大, 希望大家耐心等待, 也欢迎大家一起加入和完善.
- 如果发现不通顺或者有歧义的地方, 可以在评论里指出来, 我们会及时改正的.
- Github托管地址
- 原文地址
- 查看4.6翻译进度

如何参与&联系我们

- 邮箱: dongchuanyz@163.com
- 微信: dongchuan55
- QQ群: 324020116

任何问题都欢迎直接联系我们

Gitbook 提供了非常棒的在线编辑功能, 所以想贡献的同学可以直接联系我申请权限！

许可证

本作品采用 Apache License 2.0 国际许可协议 进行许可. 传播此文档时请注意遵循以上许可协议. 关于本许可证的更多详情可参考 http://www.apache.org/licenses/LICENSE-2.0

贡献者列表

成员	联系方式	**Github**
dongchuan55	dongchuan55@gmail.com	Github
UFreedom	sunfreedomsun@gmail.com	Github
张扬	zhangyang911120@gmail.com	Github
d0048	d0048@foxmail.com	Github

介绍

很高兴能向大家介绍 Gradle, 这是一个构建系统, 我们认为它是 java (JVM) 世界中构建技术的一个飞跃.

Gradle 提供了:

- 一个像 Ant 一样的非常灵活的通用构建工具
- 一种可切换的, 像 maven 一样的基于合约构建的框架
- 支持强大的多工程构建
- 支持强大的依赖管理(基于 Apachelvy)
- 支持已有的 maven 和 ivy 仓库
- 支持传递性依赖管理, 而不需要远程仓库或者 pom.xml 或者 ivy 配置文件
- 优先支持 Ant 式的任务和构建
- 基于 groovy 的构建脚本
- 有丰富的领域模型来描述你的构建

关于这本指南

这本用户指南还并不完善, 就像 Gradle 一样还在开发当中.

在这本指南中, Gradle 的一些功能并没有被完整的展示出来. 一些内容的解释也并不是十分的清楚, 或者假设关于 Gradle 你知道得更多. 我们需要你的帮助来完善这本指南. 在 Gradle 网站上你可以找到更多关于完善这本指南的信息.

通过这本指南, 你将会看到一些代表 Gradle 任务之间依赖关系的图表. 类似于 UML 依赖关系的表示方法, 从一个任务 A 指向另一个任务 B 的箭头代表A依赖于B.

概述

- 特点
- 为什么用 Groovy?

特点

这里简述下 Gradle 的特点.

1. 声明式构建和合约构建

Gradle 的核心是基于 Groovy 的 领域特定语言 (DSL), 具有十分优秀的扩展性. Gradle 通过提供可以随意集成的声明式语言元素将声明性构建推到了一个新的高度. 这些元素也为 Java, Groovy, OSGi, Web 和Scala 等项目提供基于合约构建的支持. 而且, 这种声明式语言是可扩展的. 你可以添加自己的语言元素或加强现有的语言元素, 从而提供简洁, 易于维护和易于理解的构建.

2. 基于依赖的编程语言

声明式语言位于通用任务图 (general purpose task graph) 的顶端，它可以被充分利用在你的构建中. 它具有强大的灵活性, 可以满足使用者对 Gradle 的一些特别的需求.

3. 让构建结构化

Gradle 的易适应性和丰富性可让你在构建里直接套用通用的设计原则. 例如, 你可以非常容易地使用一些可重用的组件来构成你的构建. 但是不必要的间接内联内容是不合适的. 不要强行拆分已经结合在一起的部分 (例如, 在你的项目层次结构中). 避免使构建难以维护. 总之, 你可以创建一个结构良好，易于维护和易于理解的构建.

4. API深化

你会非常乐意在整个构建执行的生命周期中使用 Gradle, 因为Gradle 允许你管理和定制它的配置和执行行为.

5. Gradle 扩展

Gradle 扩展得非常好. 不管是简单的独立项目还是大型的多项目构建, 它都能显著的提高效率. 这是眞正的结构构建. 顶尖水平的构建功能，还可以解决许多大公司碰到的构建 性能低下的问题.

6. 多项目构建

Gradle 对多项目的支持是非常出色的. 项目依赖是很重要的部分. 它允许你模拟在多项目构建中项目的关系，这正是你所要关注的地方. Gradle 可以适应你的项目的结构, 而不是反过来.

Gradle 提供了局部构建的功能. 如果你构建一个单独的子项目, Gradle 会构建这个子项目依赖的所有子项目. 你也可以选择依赖于另一个特别的子项目重新构建这些子项目. 这样在一些大型项目里就可以节省非常多的时间.

7. 多种方式来管理你的依赖

不同的团队有不同的管理外部依赖的方法. Gradle 对于任何管理策略都提供了合适的支持. 从远程 Maven 和 Ivy 库的依赖管理到本地文件系统的 jars 或者 dirs.

8. Gradle 是第一个构建整合工具

Ant 的 tasks是 Gradle 中很重要的部分, 更有趣是 Ant 的 projects 也是十分重要的部分. Gradle 可以直接引入Ant 项目, 并在运行时直接将 Ant targets 转换成 Gradle tasks. 你可以从 Gradle 中依赖它们, 并增强它们的功能, 甚至可以在 build.xml 文件中声明 Gradle tasks 的依赖. 并且properties, paths 等也可以通过同样的方法集成进来.

Gradle 完全支持你已有的 Maven 或者 Ivy 仓库来构造发布或者提取依赖. Gradle 也提供了一个转化器, 用来将 maven 的 pom.xml 文件转换成 Gradle 脚本. 在运行时引入 Maven 项目也会在稍后推出.

9. 易于迁移

Gradle 可以兼容任何结构. 因此你可以直接在你的产品构建的分支上开发你的 Gradle 构建, 并且二者可以并行. 我们通常建议编写一些测试代码来确保它们的功能是相同的. 通过这种方式, 在迁移的时候就不会显得那么混乱和不可靠, 这是通过婴儿学步的方式来获得最佳的实践.

10. Groovy

Gradle 的构建脚本是通过 Groovy 编写的而不是 XML. 但是并不像其他方式, 这并不是为了简单的展示用动态语言编写的原始脚本有多么强大. 不然的话, 只会导致维护构建变得非常困难. Gradle 的整个设计是朝着一种语言的方向开发的, 并不是一种死板的框架. Groovy 就像胶水一样, 把你想实现的构想和抽象的 Gradle 粘在一起. Gradle提供了一些标准的构想, 但是他们并不享有任何形式的特权. 相比于其他声明式构建系统，对我们来说这是一个比较突出的特点.

10. Gradle 包装器

Gradle 包装器允许你在没有安装 Gradle 的机器上运行 Gradle 构建. 在一些持续集成的服务器上, 这个功能将非常有用. 它同样也能降低使用一个开源项目的门槛, 也就是说构建它将会非常简单. 这个包装器对于公司来说也是很有吸引力的. 它并不需要为客户机提供相应的管理防范. 这种方式同样也能强制某一个版本 Gradle 的使用从而最小化某些支持问题.

11. 免费和开源

Gradle 是一个开源项目, 遵循 ASL 许可.

为什么用 Groovy?

我们认为在脚本构建时, 一个内部的 DSL（基于一个动态语言）相对于 XML 的优势是巨大的. 有这么多的动态语言, 为什么选择 Groovy? 答案在于 Gradle 的运行环境. 虽然 Gradle 以一个通用构建工具为核心, 但是它的重点是Java项目. 在这样的项目中, 显然团队每个成员都对 Java 非常熟悉. 我们认为构建应尽可能对所有团队成员都是透明的, 所以选择了 Groovy.

你可能会说，为什么不直接使用 Java 作为构建脚本的语言. 我们认为这是一个很有用的问题. 对于你的团队, 它要有最高的透明度和最低的学习曲线, 也就是说容易掌握. 但由于 Java 的限制, 这样的构建语言不会那么完美和强大. 而像 Python，Groovy 或 Ruby 语言用来作为构建语言会更好. 我们选择了 Groovy 是因为它给 Java 开发人员提供了迄今为止最大的透明度. 其基本的符号和类型与 Java 是一样的，其封装结构和许多其他的地方也是如此. Groovy 在这基础上提供了更多的功能, 而且与 java 有共同的基础.

对于那些同时是或者即将是 Python 或 Ruby 开发者的 Java 开发人员来说, 上述的讨论并不适用. Gradle 的设计非常适合在 JRuby 和 Jython 中创建另一个构建脚本引擎. 它对于我们来说只是目前开发中没有最高优先级. 我们十分支持任何人来做贡献, 创建额外的构建脚本引擎.

教程

接下来的教程讲先介绍Gradle的基础知识

Chapter 4, 安装 Gradle

描述如何安装 Gradle.

Chapter 5, 脚本构建基础

介绍脚本构建的基础元素: projects 和 tasks.

Chapter 6, Java 快速入门

展示如何使用 Gradle 来构建 Java 项目.

Chapter 7, 依赖管理基础

展示如何使用 Gradle 的依赖管理功能.

Chapter 8, Groovy 快速入门

展示如何使用 Gradle 来构建 Groovy 项目.

Chapter 9, 网页应用快速入门

展示如何使用 Gradle 来构建网页应用项目.

Chapter 10, 使用 Gradle 命令行

安装 Gradle

- 先决条件
- 下载
- 解压缩
- 环境变量
- 运行并测试您的安装
- JVM选项

准备阶段

Gradle 需要运行在一个 Java 环境里

- 安装一个 Java JDK 或者 JRE. 而且 Java 版本必须至少是 6 以上.
- Gradle 自带 Groovy 库, 所以没必要安装 Groovy. 任何已经安装的 Groovy 会被 Gradle 忽略.

Gradle 使用任何已经存在在你的路径中的 JDK (可以通过 **java -version** 检查, 如果有就说明系统已经安装了 Java 环境). 或者, 你也可以设置 JAVA_HOME 环境参数来指定希望使用的 JDK的安装目录.

#

下载与安装

你可以从 Gradle网站 下载任意一个已经发布的版本

解压缩

Gradle 发布的版本为 **ZIP 格式. 所有文件包含:

- Gradle 的二进制文件.
- 用户指南 (HTML 和 PDF).
- DSL参考指南.
- API文档 (Javadoc和 Groovydoc).
- 扩展的例子,包括用户指南中引用的实例，以及一些更复杂的实例来帮助用户构建自己的 build.
- 二进制源码.此代码仅供参考.

设置环境变量

然后我们需要设置环境变量

1. 添加一个 **GRADLE_HOME** 环境变量来指明 Gradle 的安装路径
2. 添加 **GRADLE_HOME/bin** 到您的 *PATH* 环境变量中. 通常, 这样已经足够运行Gradle了.

扩充教程

Max OX:

假设您下载好的 Gradle 文件在 /Users/UFreedom/gradle 目录

```
1.vim ~/.bash_profile

2.添加下面内容：
export GRADLE_HOME = /Users/UFreedom/gradle
export export PATH=$PATH:$GRADLE_HOME/bin

3.source ~/.brash_profile
```

实际配置中，您需要将上面的目录换成您的 Gradle 文件在系统中的目录.

运行并测试您的安装

然后我们需要测试 Gradle 是否已经正确安装. 您可以通过在控制台输入 **gradle** 命令来运行 Gradle. 通过 **gradle -v** 命令来检测Gradle是否已经正确安装. 如果正确安装，会输出Gradle版本信息以及本地的配置环境 (groovy 和 JVM 版本等). 显示的版本信息应该与您所下载的 gradle 版本信息相匹配.

JVM 选项

JVM 选项可以通过设置环境变量来更改. 您可以使用 GRADLE_OPTS 或者 JAVA_OPTS.

- JAVA_OPTS 是一个用于 JAVA 应用的环境变量. 一个典型的用例是在 JAVA_OPTS 里设置HTTP代理服务器(proxy),
- GRADLE_OPTS 是内存选项. 这些变量可以在 gradle 的一开始就设置或者通过 gradlew 脚本来设置.

排除故障

当使用 Gradle 时, 你肯定会碰到许多问题.

解决遇到的问题

如果你碰到了问题, 首先要确定你使用的是最新版本的 Gradle. 我们会经常发布新版本, 解决一些 bug 并加入新的功能. 所以你遇到的问题可能就在新版本里解决了.

如果你正在使用 Gradle Daemon, 先暂时关闭 daemon (你可以使用 **switch --no-daemon** 命令). 在第19章我们可以了解到更多关于 daemon 的信息.

获得帮助

可以浏览 http://forums.gradle.org 获得相关的帮助. 在 Gradle 论坛里, 你可以提交问题, 当然也可以回答其他 Gradle 开发人员和使用者的问题.

如果你碰到不能解决的问题, 请在论坛里报告或者提出这个问题, 通常这是解决问题最快的方法. 你也可以提出建议或者一些新的想法. 通过论坛, 你当然也可以获得 Gradle 最新的开发信息.

构建脚本的基础知识

这一章主要讲解以下内容

- Projects 和 tasks
- Hello world
- 快捷的任务定义
- 构建脚本代码
- 任务依赖
- 动态任务
- 使用已经存在的任务
- 快捷注释
- 附加的 task 属性
- 使用 Ant 任务
- 使用方法
- 默认的任务
- 通过 DAG 配置

Projects 和 tasks

Gradle 里的任何东西都是基于这两个基础概念:

- projects (项目)
- tasks (任务)

每一个构建都是由一个或多个 projects 构成的. 一个 project 到底代表什么取决于你想用 Gradle 做什么. 举个例子, 一个 project 可以代表一个 JAR 或者一个网页应用. 它也可能代表一个发布的 ZIP 压缩包, 这个 ZIP 可能是由许多其他项目的 JARs 构成的. 但是一个 project 不一定非要代表被构建的某个东西. 它可以代表一件**要做的事, 比如部署你的应用.

不要担心现在看不懂这些说明. Gradle 的合约构建可以让你来具体定义一个 project 到底该做什么.

每一个 project 是由一个或多个 tasks 构成的. 一个 task 代表一些更加细化的构建. 可能是编译一些 classes, 创建一个 JAR, 生成 javadoc, 或者生成某个目录的压缩文件.

目前, 我们先来看看定义构建里的一些简单的 task. 以后的章节会讲解多项目构建以及如何通过 projects 和 tasks 工作.

Hello world

你可以通过 **gradle** 命令运行一个 Gradle 构建.

gradle 命令会在当前目录中查找一个叫 **build.gradle** 的文件. 我们称这个 build.gradle 文件为一个构建脚本 (build script), 但是严格来说它是一个构建配置脚本 (build configuration script). 这个脚本定义了一个 project 和它的 tasks.

让我们来先看一个例子, 创建一个名为build.gradle的构建脚本.

例子 **6.1** 第一个构建脚本

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

在命令行里, 进入脚本所在的文件夹然后输入 **gradle -q hello** 来执行构建脚本:

gradle -q hello 的输出

```
> gradle -q hello
Hello world!
```

这里发生了什么? 这个构建脚本定义了一个独立的 task, 叫做 hello, 并且加入了一个 action. 当你运行 **gradle hello**, Gradle 执行叫做 hello 的 task, 也就是执行了你所提供的 action. 这个 action 是一个包含了一些 Groovy 代码的闭包(closure 这个概念不清楚的同学好好谷歌下).

如果你认为这些看上去和 Ant 的 targets 很相像, 好吧, 你是对的. Gradle tasks 和 Ant 的 targets 是对等的. 但是你将会会看到, Gradle tasks 更加强大. 我们使用一个不同于 Ant 的术语 task, 看上去比 target 更加能直白. 不幸的是这带来了一个术语冲突, 因为 Ant 称它的命令, 比如 javac 或者 copy, 叫 tasks. 所以当我们谈论 tasks, 是指 Gradle 的 tasks. 如果我们讨论 Ant 的 tasks (Ant 命令), 我们会直接称呼 ant task.

补充一点命令里的 **-q** 是干什么的**?**

这个指南里绝大多说的例子会在命令里加入 **-q**. 代表 quiet 模式. 它不会生成 Gradle 的日志信息 (log messages), 所以用户只能看到 tasks 的输出. 它使得的输出更加清晰. 你并不一定需要加入这个选项. 参考第 18 章, 日志的 Gradle 影响输出的详细信息.

快捷的任务定义

有一种比我们之前定义的 hello 任务更简明的方法

例子 **6.3.** 快捷的任务定义

build.gradle

```
task hello << {
    println 'Hello world!'
}
```

它定义了一个叫做 hello 的任务, 这个任务是一个可以执行的闭包. 我们将使用这种方式来定义这本指南里所有的任务.

翻译者补充

与前面的例子比较, doLast 被替换成了 <<. 它们有一样的功能, 但看上去更加简洁了, 会在后续章节 (6.7) 详细讲解它们的功能.

构建脚本代码

Gradle 的构建脚本展示了 Groovy 的所有能力. 作为开胃菜, 来看看这个:

例子 **6.4.** 在 **Gradle** 任务里使用 **Groovy**

build.gradle

```
task upper << {
    String someString = 'mY_nAmE'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
```

gradle -q upper 命令的输出

```
> gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

或者

例子 **6.5.** 在 **Gradle** 任务里使用 **Groovy**

build.gradle

```
task count << {
    4.times { print "$it " }
}
```

gradle -q count 命令的输出

```
> gradle -q count
0 1 2 3
```

任务依赖

就像你所猜想的那样, 你可以声明任务之间的依赖关系.

例子 **6.6.** 申明任务之间的依赖关系

build.gradle

```
task hello << {
    println 'Hello world!'
}

task intro(dependsOn: hello) << {
    println "I'm Gradle"
}
```

gradle -q intro 命令的输出

```
> gradle -q intro
Hello world!
I'm Gradle
```

intro 依赖于 hello, 所以执行 intro 的时候 hello 命令会被优先执行来作为启动 intro 任务的条件.

在加入一个依赖之前, 这个依赖的任务不需要提前定义, 来看下面的例子.

例子 **6.7. Lazy dependsOn -** 其他的任务还没有存在

build.gradle

```
task taskX(dependsOn: 'taskY') << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
```

gradle -q taskX 命令的输出

```
> gradle -q taskX
taskY
taskX
```

taskX 到 taskY 的依赖在 taskY 被定义之前就已经声明了. 这一点对于我们之后讲到的多任务构建是非常重要的. 任务依赖将会在 14.4 具体讨论.

请注意你不能使用快捷注释 (参考 6.8, “快捷注释”) 当所关联的任务还没有被定义.

动态任务

Groovy 不仅仅被用来定义一个任务可以做什么. 举个例子, 你可以使用它来动态的创建任务.

例子 **6.8.** 动态的创建一个任务

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
```

这里动态的创建了 task0, task1, task2, task3

gradle -q task1 命令的输出

```
> gradle -q task1
I'm task number 1
```

使用已经存在的任务

当任务创建之后, 它可以通过API来访问. 这个和 Ant 不一样. 举个例子, 你可以创建额外的依赖.

例子 **6.9.** 通过**API**访问一个任务 **-** 加入一个依赖

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
task0.dependsOn task2, task3
```

gradle -q task0 命令的输出

```
> gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

或者你可以给一个已经存在的任务加入行为.

例子 **6.10.** 通过**API**访问一个任务 **-** 加入行为

build.gradle

```
task hello << {
    println 'Hello Earth'
}
hello.doFirst {
    println 'Hello Venus'
}
hello.doLast {
    println 'Hello Mars'
}
hello << {
    println 'Hello Jupiter'
}
```

gradle -q hello 命令的输出

```
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

doFirst 和 doLast 可以被执行许多次. 他们分别可以在任务动作列表的开始和结束加入动作. 当任务执行的时候, 在动作列表里的动作将被按顺序执行. 这里第四个行为中 << 操作符是 doLast 的简单别称.

短标记法

正如同你已经在之前的示例里看到, 有一个短标记 $ 可以访问一个存在的任务. 也就是说每个任务都可以作为构建脚本的属性:

例子 **6.11.** 当成构建脚本的属性来访问一个任务

build.gradle

```
task hello << {
    println 'Hello world!'
}
hello.doLast {
    println "Greetings from the $hello.name task."
}
```

gradle -q hello 命令的输出

```
> gradle -q hello
Hello world!
Greetings from the hello task.
```

这里的 name 是任务的默认属性, 代表当前任务的名称, 这里是 hello.

这使得代码易于读取， 特别是当使用了由插件（如编译）提供的任务时尤其如此.

自定义任务属性

你可以给任务加入自定义的属性. 列如加入一个叫做 myProperty 属性, 设置一个初始值给 ext.myProperty. 然后, 该属性就可以像一个预定义的任务属性那样被读取和设置了.

例子 **6.12.** 给任务加入自定义属性

build.gradle

```
task myTask {
    ext.myProperty = "myValue"
}

task printTaskProperties << {
    println myTask.myProperty
}
```

gradle -q printTaskProperties 命令的输出

```
> gradle -q printTaskProperties
myValue
```

给任务加自定义属性是没有限制的. 你可以在 13.4.2, “自定义属性” 里获得更多的信息.

调用 Ant 任务

Ant 任务是 Gradle 的一等公民. Gradle 通过 Groovy 出色的集成了 Ant 任务. Groovy 自带了一个 AntBuilder. 相比于从一个 build.xml 文件中使用 Ant 任务, 在 Gradle 里使用 Ant 任务更为方便和强大. 从下面的例子中, 你可以学习如何执行 Ant 任务以及如何访问 ant 属性：

例子 **6.13.** 使用 **AntBuilder** 来执行 **ant.loadfile** 任务

build.gradle

```
task loadfile << {
    def files = file('../antLoadfileResources').listFiles().sort()
    files.each { File file ->
        if (file.isFile()) {
            ant.loadfile(srcFile: file, property: file.name)
            println " *** $file.name ***"
            println "${ant.properties[file.name]}"
        }
    }
}
```

gradle -q loadfile 命令的输出

```
> gradle -q loadfile
*** agile.manifesto.txt ***
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration  over contract negotiation
Responding to change over following a plan
 *** gradle.manifesto.txt ***
```

使用方法

Gradle 能很好地衡量你编写脚本的逻辑能力. 首先要做的是如何提取一个方法.

例子 **6.14.** 使用方法组织脚本逻辑

build.gradle

```
task checksum << {
    fileList('../antLoadfileResources').each {File file ->
        ant.checksum(file: file, property: "cs_$file.name")
        println "$file.name Checksum: ${ant.properties["cs_$file.name"]}"
    }
}

task loadfile << {
    fileList('../antLoadfileResources').each {File file ->
        ant.loadfile(srcFile: file, property: file.name)
        println "I'm fond of $file.name"
    }
}

File[] fileList(String dir) {
    file(dir).listFiles({file -> file.isFile() } as FileFilter).sort()
}
```

adle -q loadfile 命令的输出

```
> gradle -q loadfile
I'm fond of agile.manifesto.txt
I'm fond of gradle.manifesto.txt
```

稍后你看到, 这种方法可以在多项目构建的子项目之间共享. 如果你的构建逻辑变得更加复杂, Gradle 为你提供了其他非常方便的方法. 请参见第59章，组织构建逻辑。

默认任务

Gradle 允许在脚本中定义一个或多个默认任务.

例子 **6.15.** 定义默认任务

build.gradle

```
defaultTasks 'clean', 'run'

task clean << {
    println 'Default Cleaning!'
}

task run << {
    println 'Default Running!'
}

task other << {
    println "I'm not a default task!"
}
```

gradle -q 命令的输出

```
> gradle -q
Default Cleaning!
Default Running!
```

等价于 **gradle -q clean run**. 在一个多项目构建中, 每一个子项目都可以有它特别的默认任务. 如果一个子项目没有特别的默认任务, 父项目的默认任务将会被执行.

通过 DAG 配置

正如我们之后的详细描述 (参见第55章，构建的生命周期), Gradle 有一个配置阶段和执行阶段. 在配置阶段后, Gradle 将会知道应执行的所有任务. Gradle 为你提供一个"钩子", 以便利用这些信息. 举个例子, 判断发布的任务是否在要被执行的任务当中. 根据这一点, 你可以给一些变量指定不同的值.

在接下来的例子中, distribution 任务和 release 任务将根据变量的版本产生不同的值.

例子 **6.16.** 根据选择的任务产生不同的输出

build.gradle

```
task distribution << {
    println "We build the zip with version=$version"
}

task release(dependsOn: 'distribution') << {
    println 'We release now'
}

gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(release)) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```

gradle -q distribution 命令的输出

```
> gradle -q distribution
We build the zip with version=1.0-SNAPSHOT
Output of gradle -q release
> gradle -q release
We build the zip with version=1.0
We release now
```

最重要的是 whenReady 在 release 任务执行之前就已经影响了 release 任务. 甚至 release 任务不是首要任务 (i.e., 首要任务是指通过 gradle 命令的任务).

Java 构建入门

- Java 插件
- 一个基础的 Java 项目
- 多项目的 Java 构建

Java 插件

如你所见, Gradle 是一种多用途的构建工具. 它可以在你的构建脚本里构建任何你想要实现的东西. 但前提是你必须先在构建脚本里加入代码, 不然它什么都不会执行.

大多数 Java 项目是非常相似的: 你需要编译你的 Java 源文件, 运行一些单元测试, 同时创建一个包含你类文件的 JAR. 如果你可以不需要为每一个项目重复执行这些步骤, 我想你会非常乐意的.

幸运的是, 你现在不再需要做这些重复劳动了. Gradle 通过使用插件解决了这个问题. 插件是 Gradle 的扩展, 它会通过某种方式配置你的项目, 典型的有加入一些预配置任务. Gradle 自带了许多插件, 你也可以很简单地编写自己的插件并和其他开发者分享它. Java 插件就是一个这样的插件. 这个插件在你的项目里加入了许多任务， 这些任务会编译和单元测试你的源文件, 并且把它们都集成一个 JAR 文件里.

Java 插件是基于合约的. 这意味着插件已经给项目的许多方面定义了默认的参数, 比如 Java 源文件的位置. 如果你在项目里遵从这些合约, 你通常不需要在你的构建脚本里加入太多东西. 如果你不想要或者是你不能遵循合约, Gradle 也允许你自己定制你的项目. 事实上, 因为对 Java 项目的支持是通过插件实现的, 如果你不想要的话, 你一点也不需要使用这个插件来构建你的项目.

在后面的章节, 我们有许多机会来让你深入了解 Java 插件, 依赖管理和多项目构建. 在本章中，先来初步认识如何使用Java插件来构建一个Java项目.

一个基础的 Java 项目

让我们先来看一个简单的例子.

我们可以加入下面的代码来使用 Java 插件:

例子 **7.1.** 使用 **Java** 插件

build.gradle

```
apply plugin: 'java'
```

（注：此例子的代码可以再所有“-all”结尾的发行版的samples/java/quickstart目录下找到）

它将会把 Java 插件加入到你的项目中, 这意味着许多预定制的任务会被自动加入到你的项目里.

Gradle 默认在 **src/main/java** 目录下寻找到你的正式（生产）源码, 在 **src/test/java** 目录下寻找到你的测试源码, 并在**src/main/resources**目录下寻找到你准备打包进jar的资源文件。测试代码会被加入到环境变量中设置的目录里运行。所有的输出文件都会被创建在构建目录里, 生成的JAR文件会被存放在 **build/libs** 目录下.

都加了什么可以执行的任务呢**?**

请在项目目录下使用 **gradle tasks** 来列出该项目的所有任务。

建立项目

尽管Java 插件在你的项目里加入了许多任务，只有几个会在项目构建中经常用到。

最常用的任务是 **build** 任务, 用于完全构建你的项目.运行 **gradle build** 命令执行后,Gradle 将会编译和测试你的代码,并生成一个包含所有类与资源的 JAR 文件:

例子 **7.2.** 建立一个 **Java** 项目

gradle build 命令的输出：

```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 1 secs
```

其余一些有用的任务是:

clean

删除 **build** 目录和所有为build生成的文件.

assemble

编译并打包你的代码, 但是并不运行单元测试.其他插件会在这个任务里加入更多的步骤.举个例子,如果你使用 War 插件,这个任务还将根据你的项目生成一个 WAR 文件.

check

编译并测试你的代码. 其他的插件会加入更多的检查步骤.举个例子, 如果你使用 checkstyle 插件, 这个任务将会运行 Checkstyle 来检查你的代码.

外部的依赖

通常, 一个 Java 项目的依赖许多外部的 JAR 文件.为了在项目里引用这些 JAR 文件,你需要告诉 Gradle 去哪里找它们.在 Gradle 中,JAR 文件位于一个仓库中，这里的仓库类似于 MAVEN 的仓库，可以被用来提取依赖,或者放入依赖。

举个例子，我们将使用开放的 Maven 仓库:

例子 **7.3.** 加入 **Maven** 仓库

build.gradle

```
repositories {
    mavenCentral()
}
```

接下来让我们加入一些依赖. 这里, 我们假设我们的项目在编译阶段有一些依赖:

例子 *6.4.* 加入依赖

build.gradle

```
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

可以看到 commons-collections 被加入到了编译阶段, junit 也被加入到了测试编译阶段.

你可以在第 8 章里看到更多这方面的内容.

定制项目

Java 插件给项目加入了一些属性 (propertiy).这些属性已经被赋予了默认的值,已经足够来开始构建项目了.如果你认为不合适,改变它们的值也是很简单的.让我们看下这个例子.这里我们将指定 Java 项目的版本号,以及我们所使用的 Java 的版本.我们同样也加入了一些属性在 **jar** 的 **manifest**里.

例子 **7.5.** 定制 **MANIFEST.MF** 文件

build.gradle

```
sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Versio
n': version
    }
}
```

Java 插件加入的任务是常规性的任务,准确地说,就如同它们在构建文件里声明地一样. 这意味着你可以使用任务之前的章节提到的方法来定制这些任务.举个例子,你可以设置一个任务的属性,在任务里加入行为,改变任务的依赖, 或者完全重写一个任务,我们将配置一个测试任务,当测试执行的时候它会加入一个系统属性:

例子 **7.6.** 测试阶段加入一个系统属性

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

哪些属性是可用的**?**

你可以使用 **gradle properties** 命令来列出项目的所有属性. 这样你就可以看到 Java 插件加入的属性以及它们的默认值.

发布 **JAR** 文件

通常 JAR 文件需要在某个地方发布. 为了完成这一步, 你需要告诉 Gradle 哪里发布 JAR 文件. 在 Gradle 里, 生成的文件比如 JAR 文件将被发布到仓库里. 在我们的例子里, 我们将发布到一个本地的目录. 你也可以发布到一个或多个远程的地点.

Example 7.7. 发布 *JAR* 文件

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

运行 **gradle uploadArchives** 命令来发布 JAR 文件.

创建 Eclipse 项目

为了把你的项目导入到 Eclipse, 你需要加入另外一个插件:

Example 7.8. Eclipse 插件

build.gradle

```
apply plugin: 'eclipse'
```

现在运行 **gradle eclipse** 命令来生成 Eclipse 的项目文件. Eclipse 任务将在第 38 章, Eclipse 插件里详细讨论.

总结

下面是一个完整的构建文件的样本:

Example 7.9. Java 例子 - 完整的构建文件

build.gradle

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Versio
n': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

多项目的 Java 构建

现在让我们看一个典型的多项目构建. 下面是项目的布局:

Example 7.10. 多项目构建 - 分层布局

构建布局

```
multiproject/
  api/
  services/webservice/
  shared/
```

注意: 这个例子的代码可以在 samples/java/multiproject 里找到.

现在我们能有三个项目. 项目的应用程序接口 (API) 产生一个 JAR 文件, 这个文件将提供给用户, 给用户提供基于 XML 的网络服务. 项目的网络服务是一个网络应用, 它返回 XML. shared 目录包含被 api 和 webservice 共享的代码.

定义一个多项目构建

为了定义一个多项目构建, 你需要创建一个设置文件 (settings file). 设置文件放在源代码的根目录, 它指定要包含哪个项目. 它的名字必须叫做 **settings.gradle**. 在这个例子中, 我们使用一个简单的分层布局. 下面是对应的设置文件:

Example 7.11. 多项目构建 - settings.gradle file

settings.gradle

```
include "shared", "api", "services:webservice", "services:shared"
```

在第56章. 多项目构建, 你可以找到更多关于设置文件的信息.

通用配置

对于绝大多数多项目构建, 有一些配置对所有项目都是常见的或者说是通用的. 在我们的例子里, 我们将在根项目里定义一个这样的通用配置, 使用一种叫做配置注入的技术 (configuration injection). 这里, 根项目就像一个容器, subprojects 方法遍历这个容器的所有元素并且注入指定的配置 . 通过这种方法, 我们可以很容易的定义所有档案和通用依赖的内容清单:

Example 7.12. 多项目构建 - 通用配置

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.11'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

注意我们例子中, Java 插件被应用到了每一个子项目中plugin to each. 这意味着我们前几章看到的任务和属性都可以在子项目里被调用. 所以, 你可以通过在根目录里运行 **gradle build** 命令编译, 测试, 和 JAR 所有的项目.

项目之间的依赖

你可以在同一个构建里加入项目之间的依赖, 举个例子, 一个项目的 JAR 文件被用来编译另外一个项目. 在 api 构建文件里我们将加入一个由 shared 项目产生的 JAR 文件的依赖. 由于这个依赖, Gradle 将确保 shared 项目总是在 api 之前被构建.

Example 7.13. 多项目构建 - 项目之间的依赖

api/build.gradle

```
dependencies {
    compile project(':shared')
}
```

创建一个发行版本

(该章需加入更多内容。。。原稿写的太简单了)我们同时也加入了一个发行版本, 将会送到客户端:

Example 7.14. 多项目构建 - 发行文件

api/build.gradle

```
task dist(type: Zip) {
    dependsOn spiJar
    from 'src/dist'
    into('libs') {
        from spiJar.archivePath
        from configurations.runtime
    }
}

artifacts {
   archives dist
}
```

依赖管理的基础知识

- 什么是依赖管理
- 声明你的依赖
- 依赖配置
- 外部的依赖
- 仓库
- 发布 artifacts

什么是依赖管理?

粗略的讲, 依赖管理由两部分组成. 首先, Gradle 需要了解你的项目需要构建或运行的东西, 以便找到它们. 我们称这些传入的文件为项目的 *dependencies(依赖项)*. 其次, Gradle 需要构建并上传你的项目产生的东西. 我们称这些传出的项目文件为 *publications(发布项)*. 让我们来看看这两条的详细信息：

大多数项目都不是完全独立的. 它们需要其它项目进行编译或测试等等. 举个例子, 为了在项目中使用 Hibernate, 在编译的时候需要在 `classpath` 中添加一些 Hibernate 的 jar 路径. 要运行测试的时候, 需要在 `test classpath` 中包含一些额外的 jar, 比如特定的 JDBC 驱动或者 `Ehcache jars` .

这些传入的文件构成上述项目的依赖. Gradle 允许你告诉它项目的依赖关系, 以便找到这些依赖关系, 并在你的构建中维护它们. 依赖关系可能需要从远程的 Maven 或者 Ivy 仓库中下载, 也可能是在本地文件系统中, 或者是通过多项目构建另一个构建. 我们称这个过程为 *dependency resolution(依赖解析)*.

这一特性与 Ant 相比效率提高了许多. 使用 Ant, 你只有指定 jar 的绝对路径或相对路径才能读取 jar. 使用 Gradle, 你只需要申明依赖的名称, 然后它会通过其它的设置来决定在哪里获取这些依赖关系, 比如从 Maven 库. 你可以为 Ant 添加 Apache Ivy 库或得类似的方法, 但是 Gradle 做的更好.

通常, 一个项目本身会具有依赖性. 举个例子, 运行 Hibernate 的核心需要其他几个类库在 classpath 中. 因此, Gradle 在为你的项目运行测试的时候, 它会找到这些依赖关系, 并使其可用. 我们称之为*transitive dependencies(依赖传递)*.

大部分项目的主要目的是要建立一些文件, 在项目之外使用. 比如, 你的项目产生一个 Java 库, 你需要构建一个jar, 可能是一个 jar 和一些文档, 并将它们发布在某处.

这些传出的文件构成了项目的发布. Gradle 当然会为你负责这个重要的工作. 你声明项目的发布, Gradle 会构建并发布在某处. 究竟什么是"发布"取决于你想做什么. 可能你希望将文件复制到本地目录, 或者将它们上传到一个远程 Maven 或者 Ivy 库.或者你可以使用这些文件在多项目构建中应用在其它的项目中. 我们称这个过程为 *publication(发布)*

声明你的依赖

让我们看一下一些依赖的声明. 下面是一个基础的构建脚本:

例子 **8.1.** 声明依赖

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

这里发生了什么? 这个构建脚本声明 Hibernate core 3.6.7.最终 被用来编译项目的源代码. 言外之意是, 在运行阶段同样也需要 Hibernate core 和它的依赖. 构建脚本同样声明了需要 junit >= 4.0 的版本来编译项目测试. 它告诉 Gradle 到 Maven 中央仓库里找任何需要的依赖. 接下来的部分会具体说明.

依赖配置

在 Gradle 里, 依赖可以组合成*configurations(*配置*)*. 一个配置简单地说就是一系列的依赖. 我们称它们为*(dependency configuration)*依赖配置. 你可以使用它们声明项目的外部依赖. 正如我们将在后面看到, 它们也被用来声明项目的发布.

Java 插件定义了许多标准的配置. 下面列出了一些, 你也可以在Table 23.5, “Java 插件 - 依赖管理”里发现更多具体的信息.

compile

用来编译项目源代码的依赖.

runtime

在运行时被生成的类使用的依赖. 默认的, 也包含了编译时的依赖.

testCompile

编译测试代码的依赖. 默认的, 包含生成的类运行所需的依赖和编译源代码的依赖.

testRuntime

运行测试所需要的依赖. 默认的, 包含上面三个依赖.

各种各样的插件加入许多标准的配置. 你也可以定义你自己的配置. 参考 Section 52.3, “配置依赖”可以找到更加具体的定义和定制一个自己的依赖配置.

外部的依赖

你可以声明许多种依赖. 其中一种是*external dependency(外部依赖)*. 这是一种在当前构建之外的一种依赖, 它被存放在远程或本地的仓库里, 比如 Maven 的库, 或者 Ivy 库, 甚至是一个本地的目录.

下面的例子讲展示如何加入外部依赖

例子 *8.2.* 定义一个外部依赖

build.gradle

```
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
}
```

引用一个外部依赖需要使用 group, name 和 version 属性. 根据你想要使用的库, group 和 version 可能会有所差别.

有一种简写形式, 只使用一串字符串 `"group:name:version"` .

例子 *8.3.* 外部依赖的简写形式

build.gradle

```
dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

要了解跟多关于定义并使用依赖工作的信息,参见Section 52.4,"How to declare you dependencies".

仓库

Gradle 是怎样找到那些外部依赖的文件的呢? Gradle 会在一个*repository(仓库)*里找这些文件. 仓库其实就是文件的集合, 通过 `group` , `name` 和 `version` 整理分类. Gradle 能解析好几种不同的仓库形式, 比如 Maven 和 Ivy, 同时可以理解各种进入仓库的方法, 比如使用本地文件系统或者 HTTP.

默认地, Gradle 不提前定义任何仓库. 在使用外部依赖之前, 你需要自己至少定义一个库. 比如使用下面例子中的 Maven central 仓库:

例子 *8.4. Maven central* 仓库

build.gradle

```
repositories {
    mavenCentral()
}
```

或者使用一个远程的 Maven 仓库:

例子 *8.5.* 使用远程的 *Maven* 仓库

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

或者一个远程的 Ivy 仓库:

例子 *8.6.* 使用远程的 *Ivy* 仓库

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

你也可以使用本地的文件系统里的库. Maven 和 Ivy 都支持下载的本地.

例子 *8.7.* 使用本地的 *Ivy* 目录

build.gradle

```
repositories {
    ivy {
        // URL can refer to a local directory
        url "../local-repo"
    }
}
```

一个项目可以有好几个库. Gradle 会根据依赖定义的顺序在各个库里寻找它们, 在第一个库里找到了就不会再在第二个库里找它了.

可以在Section 50.6 章, “仓库”里找到更详细的信息.

发布 artifacts

依赖配置也可以用来发布文件[3]. 我们称这些文件 `publication artifacts` , 或者就叫 *artifacts*.

插件可以很好的定义一个项目的 artifacts, 所以你并不需要做一些特别的工作来让 Gradle 需要发布什么. 你可以通过在 uploadArchives 任务里加入仓库来完成. 下面是一个发布远程 Ivy 库的例子:

例子 **8.8.** 发布一个 **Ivy** 库

build.gradle

```
uploadArchives {
    repositories {
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "http://repo.mycompany.com"
        }
    }
}
```

现在, 当你运 `gradle uploadArchives` , Gradle 将构建和上传你的 Jar. Gradle 也会生成和上传 ivy.xml .

你也可以发布到 Maven 库.语法是稍有不同[4]. 请注意你需要加入 Maven 插件来发布一个 Maven 库. 在下面的例子里, Gradle 将生成和上传 pom.xml.

例子 *8.9.* 发布 *Maven* 库

build.gradle

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

在Chapter 53, Publishing artifacts, 发布 artifacts 里有更加具体的介绍.

> [3] 我们认为这令人困惑,我们正在在Gradle DSL中逐步的区别这两个概念.
>
> [4] 我们正在努力解决从Maven仓库发布,获取的语法一致性.

下一步?

对于依赖关系的所有细节,参见Chapter 52, 依赖管理,artifact发布细节,参见Chapter 53, Publishing artifacts.

如果你对这里提及的DSL元素感兴趣,看看Project.configurations{}), [Project.repositories{}]https://docs.gradle.org/current/dsl/org.gradle.api.Project.html#org.gradle.api.Project:repositories(groovy.lang.Closure)和[Project.dependencies{}](https://docs.gradle.org/current/dsl/org.gradle.api.Project.html#org.gradle.api.Project:dependencies(groovy.lang.Closure)).

否则,继续到其他tutorials.

Groovy 快速入门

构建 Groovy 项目时, 你需要使用 `Groovy plugin (Groovy插件)` . 这个插件扩展了 Java 插件, 加入了编译 Groovy 的依赖. 你的项目可以包含 Groovy 的源代码, Java 源代码, 或者它们的混合. 在其他方面,一个Groovy项目与Java项目是相同的,就像我们在Chapter 7, Java Quickstart见到的一样.

一个基本的 Groovy 项目

让我们看一个例子. 为了使用 Groovy 插件, 加入下面的代码:

例子 **8.1. Groovy** 插件

build.gradle

```
apply plugin: 'groovy'
```

> 注意:这个例子的代码可以在 `samples/groovy/quickstart` 在Gradle分布的 `"-all"` 中找到.

它也会同时把 Java 插件加入到你的项目里. Groovy 插件扩展了编译任务, 这个任务会在 src/main/groovy 目录里寻找源代码文件, 并且加入了编译测试任务来寻找 src/test/groovy 目录里的测试源代码. 编译任务使用 联合编译 (joint compilation) 来编译这些目录, 这里的联合指的是它们混合有 java 和 groovy 的源文件.

使用 groovy 编译任务, 你必须声明 Groovy 的版本和 Groovy 库的位置. 你可以在配置文件里加入依赖, 编译配置会继承这个依赖, 然后 groovy 库将被包含在 classpath 里.

例子 *8.2. Groovy 2.2.0*

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.3'
}
```

下面是完整的构建文件:

例子 *8.3.* 完整的构建文件

build.gradle

```
apply plugin: 'eclipse'
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.3'
    testCompile 'junit:junit:4.11'
}
```

运行 **gradle build** 命令将会开始编译, 测试和创建 JAR 文件.

总结

这一章描述了一个非常简单的 Groovy 项目. 通常, 一个真正的项目要比这个复杂的多. 因为 Groovy 项目是一个 Java 项目, 任何你可以对 Java 项目做的配置也可以对 Groovy 项目做.

Chapter 24,The Groovy Plugin有更加详细的描述, 你也可以在 samples/groovy 目录里找到更多的例子.

网页应用快速入门

> 本章是一项正在进行中的工作.

Gradle 提供了两个插件用来支持网页应用: War 插件和 Jetty 插件. War 插件是在 Java 插件的基础上扩充的用来构建 WAR 文件. Jetty 插件是在 War 插件的基础上扩充的, 允许用户将网页应用发布到一个介入的 Jetty 容器里.

构建一个 WAR 文件

为了构建一个 WAR 文件, 需要在项目中加入 War 插件:

例子 *9.1. War* 插件

build.gradle

```
apply plugin: 'war'
```

> 注意:项目代码可以在 `samples/webApplication/quickstart` 在Gradle的发行包 `"-all"` 中找到.

这个插件也会在你的项目里加入 Java 插件. 运行 **gradle build** 将会编译, 测试和创建项目的 WAR 文件. Gradle 将会把源文件包含在 WAR 文件的 src/main/webapp 目录里. 编译后的 classe , 和它们运行所需要的依赖也会被包含在 WAR 文件里.

Running your web application

要启动Web工程,在项目中加入Jetty plugin即可:

例 *9.2. 采用Jetty plugin启动web工程*

build.gradle

```
apply plugin: 'jetty'
```

由于Jetty plugin继承自War plugin.使用 `gradle jettyRun` 命令将会把你的工程启动部署到jetty容器中. 调用 `gradle jettyRunWar` 命令会打包并启动部署到jetty容器中.

> **Groovy web applications** 你可以结合多个插件在一个项目中,所以你可以一起使用 `War` 和 `Groovy` 插件构建一个 Groovy 基础 web 应用.合适的 Groovy 类库会添加到你的 `WAR` 文件中.

TODO: which url, configure port, uses source files in place and can edit your files and reload.

总结

了解更多关于War plugin和Jetty plugin的应用请参阅Chapter 26,The War Plugin以及Chapter 28,The Jetty Plugin.

你可以在发行包的samples/webApplication下找到更多示例.

使用 **Gradle** 命令行

本章介绍了命令行的一些基本功能.正如在前面的章节里你所见到的调用 **gradle** 命令来运行一个构建.

多任务调用

你可以以列表的形式在命令行中一次调用多个任务. 例如 **gradle compile test** 命令会依次调用 compile 和 test 任务, 它们所依赖的任务也会被调用. 这些任务只会被调用一次, 无论它们是否被包含在脚本中:即无论是以命令行的形式定义的任务还是依赖于其它任务都会被调用执行. 来看下面的例子.

下面定义了四个任务 dist和test 都 依赖于 compile ,只用当 compile 被调用之后才会调用 `gradle dist test` 任务

示例图 **11.1.** 任务依赖

例子 **11.1.** 多任务调用

build.gradle

```
task compile << {
    println 'compiling source'
}

task compileTest(dependsOn: compile) << {
    println 'compiling unit tests'
}

task test(dependsOn: [compile, compileTest]) << {
    println 'running unit tests'
}

task dist(dependsOn: [compile, test]) << {
    println 'building the distribution'
}
```

gradle dist test 命令的输出

```
> gradle dist test
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

由于每个任务仅会被调用一次,所以调用**gradle test test**与调用**gradle test**效果是相同的.

排除任务

你可以用命令行选项-x来排除某些任务,让我们用上面的例子来示范一下.

例子 *11.2.* 排除任务

`gradle dist -x test` 命令的输出

```
> gradle dist -x test
:compile
compiling source
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

可以看到, test 任务并没有被调用,即使它是 dist 任务的依赖. 同时 test 任务的依赖任务 compileTest 也没有被调用,而像 compile 被 test 和其它任务同时依赖的任务仍然会被调用.

失败后继续执行构建

默认情况下, 只要有任务调用失败, Gradle就会中断执行. 这可能会使调用过程更快, 但那些后面隐藏的错误就没有办法发现了. 所以你可以使用 `--continue` 选项在一次调用中尽可能多的发现所有问题.

采用 `--continue` 选项, Gralde 会调用每一个任务以及它们依赖的任务. 而不是一旦出现错误就会中断执行.所有错误信息都会在最后被列出来.

一旦某个任务执行失败,那么所有依赖于该任务的子任务都不会被调用.例如由于 test 任务依赖于 complie 任务,所以如果 compile 调用出错, test 便不会被直接或间接调用.

简化任务名

当你试图调用某个任务的时候, 你并不需要输入任务的全名. 只需提供足够的可以唯一区分出该任务的字符即可. 例如, 上面的例子你也可以这么写. 用 `gradle di` 来直接调用 dist 任务:

例 **11.3.** 简化任务名

`gradle di` 命令的输出

```
> gradle di
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

你也可以用在驼峰命名方式 (通俗的说就是每个单词的第一个字母大写,除了第一个单词) 的任务中每个单词的首字母进行调用. 例如, 可以执行 `gradle compTest` 或者 `gradle cT` 来调用 compileTest 任务

例 **11.4.** 简化驼峰方式的任务名

`gradle cT` 命令的输出

```
> gradle cT
:compile
compiling source
:compileTest
compiling unit tests

BUILD SUCCESSFUL

Total time: 1 secs
```

简化后你仍然可以使用-x参数.

选择执行构建

调用 gradle 命令时, 默认情况下总是会构建当前目录下的文件, 可以使用 **-b** 参数选择其他目录的构建文件, 并且当你使用此参数时 settings.gradle 将不会生效, 看下面的例子:

例 **11.5.** 选择文件构建

subdir/myproject.gradle

```
task hello << {
    println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
}
```

gradle -q -b subdir/myproject.gradle hello 的输出

```
gradle -q -b subdir/myproject.gradle hello
using build file 'myproject.gradle' in 'subdir'.
```

另外,你可以使用 **-p** 参数来指定构建的目录,例如在多项目构建中你可以用 -p 来替代 -b 参数

例 10.6. 选择构建目录

gradle -q -p subdir hello 命令的输出

```
gradle -q -p subdir hello
using build file 'build.gradle' in 'subdir'.
```

-b 参数用以指定脚本具体所在位置, 格式为 dirpwd/build.gradle.

-p 参数用以指定脚本目录即可.

获取构建信息

Gradle提供了许多内置任务来收集构建信息. 这些内置任务对于了解依赖结构以及解决问题都是很有帮助的.

了解更多, 可以参阅项目报告插件以为你的项目添加构建报告

项目列表

执行 `gradle projects` 命令会为你列出子项目名称列表.

例 **11.7.** 收集项目信息

gradle -q projects 命令的输出结果

```
> gradle -q projects
------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

这份报告展示了每个项目的描述信息. 当然你可以在项目中用 **description** 属性来指定这些描述信息.

例 **10.8.** 为项目添加描述信息

build.gradle

```
description = 'The shared API for the application'
```

任务列表

执行 `gradle tasks` 命令会列出项目中所有任务. 这会显示项目中所有的默认任务以及每个任务的描述.

例 11.9 获取任务信息

gradle -q tasks 命令的输出

```
> gradle -q tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
dependencies - Displays all dependencies declared in root project 'projectReports'.
dependencyInsight - Displays the insight into a specific dependency in root project 'p
rojectReports'.
help - Displays a help message
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the di
splayed tasks may belong to subprojects).

To see all tasks and more detail, run with --all.
```

默认情况下,这只会显示那些被分组的任务. 你可以通过为任务设置 **group** 属性和 **description** 来把 这些信息展示到结果中.

例 **11.10.** 更改任务报告内容

build.gradle

```
dists {
    description = 'Builds the distribution'
    group = 'build'
}
```

当然你也可以用 **--all** 参数来收集更多任务信息. 这会列出项目中所有任务以及任务之间的依赖关系.

例 11.11 获得更多的任务信息

gradle -q tasks --all 命令的输出

```
> gradle -q tasks --all
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
api:clean - Deletes the build directory (build)
webapp:clean - Deletes the build directory (build)
dists - Builds the distribution [api:libs, webapp:libs]
    docs - Builds the documentation
api:libs - Builds the JAR
    api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
    webapp:compile - Compiles the source files

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
dependencies - Displays all dependencies declared in root project 'projectReports'.
api:dependencies - Displays all dependencies declared in project ':api'.
webapp:dependencies - Displays all dependencies declared in project ':webapp'.
dependencyInsight - Displays the insight into a specific dependency in root project 'p
rojectReports'.
api:dependencyInsight - Displays the insight into a specific dependency in project ':a
pi'.
webapp:dependencyInsight - Displays the insight into a specific dependency in project
':webapp'.
help - Displays a help message
api:help - Displays a help message
webapp:help - Displays a help message
projects - Displays the sub-projects of root project 'projectReports'.
api:projects - Displays the sub-projects of project ':api'.
webapp:projects - Displays the sub-projects of project ':webapp'.
properties - Displays the properties of root project 'projectReports'.
api:properties - Displays the properties of project ':api'.
webapp:properties - Displays the properties of project ':webapp'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the di
splayed tasks may belong to subprojects).
api:tasks - Displays the tasks runnable from project ':api'.
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

获取任务具体信息

执行 **gradle help --task someTask** 可以显示指定任务的详细信息. 或者多项目构建中相同任务名称的所有任务的信息. 如下例.

例 *11.12.* 获取任务帮助

*gradle -q help --task libs*的输出结果

```
> gradle -q help --task libs
Detailed task information for libs

Paths
     :api:libs
     :webapp:libs

Type
     Task (org.gradle.api.Task)

Description
     Builds the JAR
```

这些结果包含了任务的路径、类型以及描述信息等.

获取依赖列表

执行 `gradle dependencies` 命令会列出项目的依赖列表, 所有依赖会根据任务区分,以树型结构展示出来. 如下例:

例 **11.13.** 获取依赖信息

gradle -q dependencies api:dependencies webapp:dependencies 的输出结果

```
> gradle -q dependencies api:dependencies webapp:dependencies
------------------------------------------------------------
Root project
------------------------------------------------------------

No configurations

------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

compile
\--- org.codehaus.groovy:groovy-all:2.3.3

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3

------------------------------------------------------------
Project :webapp - The Web application implementation
------------------------------------------------------------

compile
+--- project :api
|    \--- org.codehaus.groovy:groovy-all:2.3.3
\--- commons-io:commons-io:1.2

testCompile
No dependencies
```

虽然输出结果很多, 但这对于了解构建任务十分有用, 当然你可以通过 `--configuration` 参数来查看 指定构建任务的依赖情况:

例 **11.14.** 过滤依赖信息

`gradle -q api:dependencies --configuration testCompile` 的输出结果

```
> gradle -q api:dependencies --configuration testCompile
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3
```

查看特定依赖

执行 `gradle dependencyInsight` 命令可以查看指定的依赖. 如下面的例子.

例 **11.15.** 获取特定依赖

`gradle -q webapp:dependencyInsight --dependency groovy --configuration compile` 的输出结果

```
> gradle -q webapp:dependencyInsight --dependency groovy --configuration compile
org.codehaus.groovy:groovy-all:2.3.3
\--- project :api
     \--- compile
```

这个 task 对于了解依赖关系、了解为何选择此版本作为依赖十分有用.了解更多请参阅 DependencyInsightReportTask.

dependencyInsight 任务是'Help'任务组中的一个. 这项任务需要进行配置才可以使用. Report 查找那些在指定的配置里特定的依赖.

如果用了Java相关的插件, 那么 dependencyInsight 任务已经预先被配置到'compile'下了. 你只需要通过 `'--dependency'` 参数来制定所需查看的依赖即可. 如果你不想用默认配置的参数项你可以通过 `'--configuration'` 参数来进行指定.了解更多请参阅 DependencyInsightReportTask.

获取项目属性列表

执行 `gradle properties` 可以获取项目所有属性列表. 如下例:

例 **11.16.** 属性信息

`gradle -q api:properties` 的输出结果

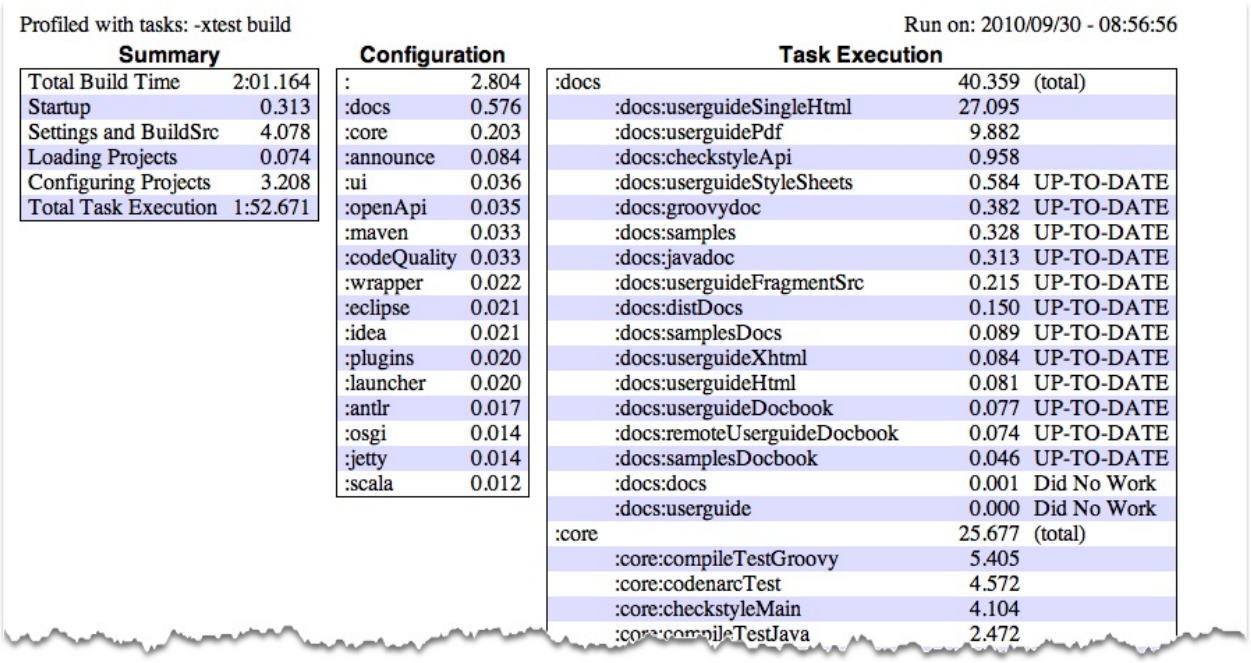
```
> gradle -q api:properties
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler@12345
asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@12345
baseClassLoaderScope: org.gradle.api.internal.initialization.DefaultClassLoaderScope@1
2345
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build.gradl
e
```

构建日志

--profile 参数可以收集一些构建期间的信息并保存到 build/reports/profile 目录下. 并且会以构建时间命名这些文件.

下面是一份日志. 这份日志记录了总体花费时间以及各过程花费的时间. 并以时间大小倒序排列. 并且记录了任务的执行情况.

如果采用了 buildSrc, 那么在 buildSrc/build 下同时也会生成一份日志记录记录.



Profiled with tasks: -xtest build

Run on: 2010/09/30 - 08:56:56

Summary

Total Build Time	2:01.164
Startup	0.313
Settings and BuildSrc	4.078
Loading Projects	0.074
Configuring Projects	3.208
Total Task Execution	1:52.671

Configuration

:	2.804
:docs	0.576
:core	0.203
:announce	0.084
:ui	0.036
:openApi	0.035
:maven	0.033
:codeQuality	0.033
:wrapper	0.022
:eclipse	0.021
:idea	0.021
:plugins	0.020
:launcher	0.020
:antlr	0.017
:osgi	0.014
:jetty	0.014
:scala	0.012

Task Execution

:docs	40.359	(total)
:docs:userguideSingleHtml	27.095	
:docs:userguidePdf	9.882	
:docs:checkstyleApi	0.958	
:docs:userguideStyleSheets	0.584	UP-TO-DATE
:docs:groovydoc	0.382	UP-TO-DATE
:docs:samples	0.328	UP-TO-DATE
:docs:javadoc	0.313	UP-TO-DATE
:docs:userguideFragmentSrc	0.215	UP-TO-DATE
:docs:distDocs	0.150	UP-TO-DATE
:docs:samplesDocs	0.089	UP-TO-DATE
:docs:userguideXhtml	0.084	UP-TO-DATE
:docs:userguideHtml	0.081	UP-TO-DATE
:docs:userguideDocbook	0.077	UP-TO-DATE
:docs:remoteUserguideDocbook	0.074	UP-TO-DATE
:docs:samplesDocbook	0.046	UP-TO-DATE
:docs:docs	0.001	Did No Work
:docs:userguide	0.000	Did No Work
:core	25.677	(total)
:core:compileTestGroovy	5.405	
:core:codenarcTest	4.572	
:core:checkstyleMain	4.104	
:core:compileTestJava	2.472	

使用 Gradle 图形界面

为了辅助传统的命令行交互，Gradle还提供了一个图形界面.我们可以使用Gradle命令中**--gui**选项来启动它.

例子 12.1. 启动图形界面

```
gradle --gui
```

注意：这个命令执行后会使得命令行一直处于封锁状态，直到我们关闭图形界面. 不过我们可以另外加上**“&”**让它在后台执行：

```
gradle --gui&
```

如果我们从自己的Gradle项目目录中启动这个图形界面，我们应该会看到任务树.

图 12.1. 任务树

我们建议您从当前的Gradle项目目录启动图形界面，因为这种方式可以将有关于界面的一些设置存储到您目录里面.不过您也可以在启动它后切换工作目录，方式：通过界面中**“Setup”**选项卡可以设置.

如您所见，这个界面在顶部有4个选项卡和底部一个输出窗口.

任务树

任务树使用分层方式显示了所有的项目和它们的任务，双击一个任务，您就可以执行它.

另外我们还可以使用过滤器过滤掉不常用的任务. 您可以点击 **Filter** 按钮来设置过滤条件. 设定哪些任务和项目可以显示. 隐藏的任务会使用红色来标记.

注意：最新被创建的任务会默认被显示出来(相反是被隐藏).

如您所见，在任务树界面我们可以做以下几种事情：

- 执行任务时忽略依赖性，而且并不需要重新编译独立的项目.
- 添加自己喜欢的任务，将其收藏(具体请看**“Favorities”**选项卡).
- 隐藏自己不想看到的任务，这个操作会将他们添加到过滤器中.
- 编辑 build.gradke 文件， 注意：这个需要你的jdk版本为1.6以上，而且你的操作系统需要关联 .gradle 文件.

收藏夹

"Favorites"选项卡是个好地方. 您可以收藏常用的命令. 即使是复杂的命令集，只要它符合**Gradle**规范，您都可以添加收藏，而且您还可以为它起个通俗易懂的别名. 这个方法逼格是不是很高. 一个一眼看上去就让人明白的自定义的命令，我们可以称它为“侩子手”（fast build）.

您还可以对收藏的任务进行排序，或者您可以导出它们到磁盘，然后将导出的命令分享给别的小伙伴使用. 如果您想编辑它们，如您所见，我们会看到有个**"Always Show Live Output"**选项，如果您勾选了，可以强制命令在执行时显示在输出窗口.

命令行

在**“Command Line”**选项卡，您只需将命令填入到**gradle**输入框. 就可以直接执行单个的Gradle命令. 或者说在您将某个命令添加到收藏夹之前，您想看看是什么效果的话，不妨来这里试试.

设置

在设置界面，你可以配置一些常用的设置.

图 12.2 设置界面

- **“Current Directory”** 图形界面会默认设置您的Gradle项目的根目录(build.gradle 文件所在的目录)为当前目录.
- **“Stack Trace Output“** 这个选项可以指定当错误发生时，有多少信息可以写入到轨迹栈中，注意：在您设定轨迹栈级别后，如果"Command Line"(命令行)选项卡中，或者在"Favorites"（收藏夹）选项卡中的命令发生错误,这个设置就不会起作用了.
- **”Only Show Output When Errors Occur”** 设定当编译出问题时输出窗口才显示相关信息.
- **"Use Custom Gradle Executor"** 高级功能，您可以指定一个路径启动Gradle命令代替默认的设置，例如您的项目需要在别的批处理文件或者shell脚本进行额外的配置（例如指定一个初始化脚本），这种情况您就可以使用它.

编写构建脚本

这一章我们将要深入的学习如何编写构建脚本.

Gradle 构建语言

Gradle 是以 Groovy 语言为基础, 基于DSL (领域特定语言) 语法的自动化构建工具,但是它增加了一些额外的特性,这使得Gradle更加的容易去阐释构建.

一个构建脚本能够包含任何Groovy语言的元素 (Any language element except for statement labels), 每个构建脚本都使用UTF-8编码.

项目 API

在第七章 Java构建入门那部分我们使用了 apply() 方法，这个方法是从哪里来的呢? 我们之前说过Gradle在构建脚本中定义了一个项目. 对于构建脚本中每个项目，Gradle 都创建了一个 Project 类型的对象用来关联此项目. 当构建脚本执行时，它会去配置所关联的工程对象.

- 构建脚本中每个被调用的方法（这些方法并未在构建脚本中定义）都被委托给当前工程对象（使用工程对象引用方法）。
- 构建脚本中每个被操作的属性（这些属性并未在构建脚本中定义）都被委托给当前工程对象（使用工程对象引用属性）.

让我们尝试下如何操作工程对象的属性.

例子：13.1 操作工程对象的属性

build.gradle

```
println name
println project.name
```

使用 **gradle -q check** 命令输出结果:

```
> gradle -q check
projectApi
projectApi
```

如您所见，两个 **println** 语句都输出了相同的属性，第一个输出使用的是自动委托 (auto-delegation), 因为当前属性并没有在构建脚本中定义. 另一个语句使用了项目一个属性，这个属性在任何构建脚本中都可用，它的返回值是被关联的工程对象. 只有当您定义了一个属性或者一个方法, 它的名字和工程对象的某个成员的名字相同时, 你应该使用项目属性.

标准项目属性

Project 对象提供了一些标准的属性，您可以在构建脚本中很方便的使用他们. 下面列出了常用的属性:

Name	Type	Default Value
project	Project	Project 实例对象
name	String	项目目录的名称
path	String	项目的绝对路径
description	String	项目描述
projectDir	File	包含构建脚本的目录
build	File	*projectDir*/build
group	Object	未具体说明
version	Object	未具体说明
ant	AntBuilder	Ant实例对象

建议：

不要忘记我们的构建脚本只是个很简单的 Groovy 代码 ，不过它会再调用 **Gradle API**，Project 接口通过调用 **Gradle API** 让我们可以操作任何事情，因此如果你想知道哪个标签('tags') 可以在构建脚本种使用，您可以翻阅 Project 接口的的说明文档.

脚本 API

当 Gradle 执行一个脚本时，它会将这个脚本编译为实现了 Script 的类. 也就是说所有的属性和方法都是在 Script 接口中声明的，由于你的脚本实现了 Script 接口，所以你可以在自己的脚本中使用它们.

声明变量

在 Gradle 构建脚本中有两种类型的变量可以声明：局部变量 (local) 和 扩展属性 (extra) .

局部变量

局部变量使用关键字 def 来声明，其只在声明它的地方可见 . 局部变量是 Groovy 语言的一个基本特性.

例子 **13.2 .** 使用局部变量

```
def dest = "dest"

task copy(type: Copy) {
      form "source"
      into dest

}
```

扩展属性

在 Gradle 领域模型中所有被增强的对象能够拥有自己定义的属性. 这包括，但不仅限于 projects , tasks , 还有 source sets . Project 对象可以添加，读取，更改扩展的属性. 另外，使用 ext 扩展块可以一次添加多个属性.

例子 **13.3.** 使用扩展属性

build.gradle

```
apply plugin: "java"

ext {
    springVersion = "3.1.0.RELEASE"
    emailNotification = "build@master.org"
}

sourceSets.all { ext.purpose = null }

sourceSets {
    main {
        purpose = "production"
    }
    test {
        purpose = "test"
        }
    plugin {
        purpose = "production"
    }

    }

    task printProperties << {
        println springVersion
        println emailNotification
        sourceSets.matching { it.purpose == "production" }.each { println it.name }
    }
```

使用**gradle -q printProperties**输出结果

```
> gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin
```

在上面的例子中，一个 ext 扩展块向 Project 对象添加了两个扩展属性. 名为 perpose 的属性被添加到每个 source set，然后设置 ext.purpose 等于 null (null值是被允许的). 当这些扩展属性被添加后，它们就像预定义的属性一样可以被读取，更改值.

例子中我们通过一个特殊的语句添加扩展属性，当您试图设置一个预定义属性或者扩展属性，但是属性名拼写错误或者并不存在时，操作就会失败. Project 对象可以在任何地方使用其扩展属性 ，它们比局部变量有更大的作用域. 一个项目的扩展属性对其子项目也可见.

关于扩展属性更多的细节还有它的API，请看 ExtraPropertiesExtension 类的 API 文档说明.

Groovy 基础

Groovy 提供了大量的特性用来创建 DSL. Gradle 构建语言知道 Groovy 语言的工作原理，并利用这些特性帮助您编写构建脚本，特别是您在编写 plugin 或者 task 的时候，你会觉得很方便.

Groovy JDK

Groovy 在 Java 基础上添加了很多有用的方法. 例如，Iterable 有一个 each 方法， 通过使用 each 方法,我们可以迭代出 Iterable 中的每一个元素:

例子**: 13.4.Groovy JDK** 方法

build.gradle

```
configuration.runtime.each { File f -> println f }
```

更多内容请阅读 http://groovy.codehaus.org/groovy-jdk/

属性存取器

Groovy 自动将一个属性的引用转换为相应的 getter 或 setter 方法.

例子**: 13.5.** 属性存取器

```
// 使用 getter 方法
println project.buildDir
println getProject().getBuildDir()

// 使用 setter 方法
project.buildDir = 'target'
getProject().setBuildDir('target')
```

可有可无的圆括号

在调用方法时，圆括号可有可无，是个可选的.

例子**: 13.6.**不使用圆括号调用方法

build.gradle

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

List 和 Map 集合

Groovy 为预定义的 List 和 Map 集合提供了一些操作捷径，这两个字面值都比较简单易懂，但是 Map 会有一些不同.

例如，当您使用 "apply" 方法使用插件时，apply 会自动加上 Map 的一个参数，当您这样写 " apply plugin: 'java' "时，实际上使用的是 name 参数(name-value)，只不过在 Groovy 中 使用 Map 没有 < > ,当方法被调用的时候，name 参数就会被转换成 Map 键值对，只不过在 Groovy 中看起来不像一个 Map.

**例子 13.7.List 和 Map 集合

build.gradle

```
// List 集合
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']

List<String> list = new ArraryList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list

// Map 集合
Map<String,String> map = [key1:'value1', key2:'valu2']

// Groovy 会强制将Map的键值对转换为只有value的映射
apply plugin: 'java'
```

闭包作为方法的最后一个参数

Gradle DSL 在很多地方使用闭包，这里我们将讨论更多关于闭包的使用. 当一个方法的最后一个参数是一个闭包时，您可以在方法调用后放置一个闭包.

例子**: 13.8.**闭包作为方法的参数

build.gradle

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({println "in a closure" })
```

闭包委托对象

每个闭包都有一个委托对象，当闭包既不是局部变量也不是作为方法参数时，Groovy 使用委托对象查找变量和方法引用. 当委托对象被用来管理时，Gradle 使用它来管理闭包.

例子 **13.9.**闭包引用

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testCompile('junit:junit:4.11')
    delegate.testCompile('junit:junit:4.11')
}
```

深入了解 Tasks

在这本教程的一开始 (第 6 章, 构建脚本基础) 你已经学习了如何创建简单的任务. 然后你也学习了如何给这些任务加入额外的行为, 以及如何在任务之间建立依赖关系. 这些仅仅是用来构建简单的任务. Gradle 可以创建更为强大复杂的任务. 这些任务可以有它们自己的属性和方法. 这一点正是和 Ant targets 不一样的地方. 这些强大的任务既可以由你自己创建也可以使用 Gradle 内建好的.

定义 tasks

我们已经在第 6 章学习了定义任务的形式 (keyword 形式). 当然也会有一些定义形式的变化来适应某些特殊的情况. 比如下面的例子中任务名被括号括起来了. 这是因为之前定义简单任务的形式 (keyword 形式) 在表达式里是不起作用的.

例子 **15.1.** 定义 **tasks**

build.gradle

```
task(hello) << {
    println "hello"
}

task(copy, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

你也可以使用 strings 来定义任务的名字:

例子 **15.2.** 例子 **tasks -** 使用 **strings** 来定义任务的名字

build.gradle

```
task('hello') <<
{
    println "hello"
}

task('copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

还有另外一种语法形式来定义任务, 更加直观:

例子 **15.3.** 另外一种语法形式

build.gradle

```
tasks.create(name: 'hello') << {
    println "hello"
}

tasks.create(name: 'copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

这里实际上我们把任务加入到 tasks collection 中. 可以看一看 TaskContainer 来深入了解下.

定位 tasks

你经常需要在构建文件里找到你定义的 tasks, 举个例子, 为了配置它们或者使用它们作为依赖. 有许多种方式都可以来实现定位. 首先, 每一个任务都必须是一个 project 的有效属性, 并使用任务名来作为属性名:

例子 **15.4.** 通过属性获取 **tasks**

build.gradle

```
task hello

println hello.name
println project.hello.name
```

Tasks 也可以通过 tasks collection 来得到.

例子 **15.5.** 通过 **tasks collection** 获取 **tasks**

build.gradle

```
task hello

println tasks.hello.name
println tasks['hello'].name
```

你也可以使用 tasks.getByPath() 方法通过任务的路径来使用任何 project 里的任务. 你可以通过使用任务的名字, 任务的相对路径或者绝对路径作为 getByPath() 方法的输入.

例子 **15.6.** 通过路径获取 **tasks**

build.gradle

```
project(':projectA') {
    task hello
}

task hello

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

gradle -q hello 的输出

```
> gradle -q hello
:hello
:hello
:projectA:hello
:projectA:hello
```

参考 TaskContainer 可以知道跟多关于定位 tasks 的选项.

配置 tasks

举一个例子, 让我们看一看 Gradle 自带的 Copy task. 为了创建一个 Copy task, 你需要在你的构建脚本里先声明它:

例子 **15.7.** 创建一个 **copy task**

build.gradle

```
task myCopy(type: Copy)
```

它创建了一个没有默认行为的 copy task. 这个 task 可以通过它的 API 来配置(参考 Copy). 接下来例子展示了不同的实现方法.

补充说明一下, 这个 task 的名字是 “myCopy”, 但是它是 “Copy” 类(type). 你可以有许多同样 type 不同名字的 tasks. 这个在实现特定类型的所有任务的 cross-cutting concerns 时特别有用.

例子 **15.8.** 配置一个任务 **-** 不同的方法

build.gradle

```
Copy myCopy = task(myCopy, type: Copy)
myCopy.from 'resources'
myCopy.into 'target'
myCopy.include('**/*.txt', '**/*.xml', '**/*.properties')
```

这个我们通过 Java 配置对象是一样的形式. 但是你每次都必须在语句里重复上下文 (myCopy). 这种方式可能读起来并不是那么的漂亮.

下面一种方式就解决了这个问题. 是公认的最具可读性的方式.

例子 **15.9.** 配置一个任务 **-** 通过闭包 **closure**

build.gradle

```
task myCopy(type: Copy)

myCopy {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

上面例子里的第三行是 tasks.getByName() 方法的一个简洁的写法. 特别要注意的是, 如果你通过闭包的形式来实现 getByName() 方法, 这个闭包会在 task 配置的时候执行而不是在 task 运行的时候执行.

你也可以直接在定义 task 时使用闭包.

例子 **15.10.** 通过定义一个任务

```
build.gradle

task copy(type: Copy) {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

请不要忘了构建的各个阶段.

一个任务有配置和动作. 当使用 << 时, 你只是简单的使用捷径定义了动作. 定义在配置区域的代码只会在构建的配置阶段执行, 而且不论执行哪个任务. 可以参考第 55 章, The Build Lifecycle for more details about the build lifecycle.

给 task 加入依赖

有许多种加入依赖的方式. 在 6.5 小节, “任务依赖”里, 你已经学习了如何使用任务的名称定义依赖. 任务名称可以指向同一个项目里的任务, 或者其他项目里的任务. 为了指向其他项目, 你必须在任务的名称前加入项目的路径. 下面的例子给 projectA:taskX 加入依赖 projectB:taskY :

例子 **15.11.** 从另外一个项目给任务加入依赖

build.gradle

```
project('projectA') {
    task taskX(dependsOn: ':projectB:taskY') << {
        println 'taskX'
    }
}

project('projectB') {
    task taskY << {
        println 'taskY'
    }
}
```

gradle -q taskX 的输出

```
> gradle -q taskX
taskY
taskX
```

除了使用任务名称, 你也可以定义一个依赖对象y:

例子 **15.12.** 通过任务对象加入依赖

build.gradle

```
task taskX << {
    println 'taskX'
}

task taskY << {
    println 'taskY'
}

taskX.dependsOn taskY
```

gradle -q taskX 的输出

```
> gradle -q taskX
taskY
taskX
```

更加先进的用法, 你可以通过闭包定义一个任务依赖. 闭包只能返回一个单独的 Task 或者 Task 对象的 collection, 这些返回的任务就将被当做依赖. 接下来的例子给 taskX 加入了一个复杂的依赖, 所有以 lib 开头的任务都将在 taskX 之前执行:

例子 **15.13.** 通过闭包加入依赖

build.gradle

```

task taskX << {
    println 'taskX'
}

taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

task notALib << {
    println 'notALib'
}
```

gradle -q taskX 的输出

```
> gradle -q taskX
lib1
lib2
taskX
```

For more information about task dependencies, see the Task API.

给 **tasks** 排序

> 任务的排序功能正在测试和优化. 请注意, 这项功能在 Gradle 之后的版本里可能会改变.

在某些情况下, 我们希望能控制任务的的执行顺序, 这种控制并不是向上一张那样去显示地加入依赖关系. 最主要的区别是我们设定的排序规则不会影响那些要被执行的任务, 只是影响执行的顺序本身. 好吧, 我知道可能有点抽象.

我们来看看以下几种有用的场景:

- 执行连续的任务: eg. 'build' 从来不会在 'clean' 之前执行.
- 在 build 的一开始先运行构建确认 (build validations): eg. 在正式的发布构建前先确认我的证书是正确的.
- 在运行长时间的检测任务前先运行快速的检测任务来获得更快的反馈: eg. 单元测试总是应该在集成测试之前被执行.
- 一个聚集 (aggregates) 某种特定类型的所有任务结果的任务: eg. 测试报告任务 (test report task) 包含了所有测试任务的运行结果.

目前, 有 2 种可用的排序规则: **"must run after"** 和 **"should run after"**.

当你使用 “must run after” 时即意味着 taskB 必须总是在 taskA 之后运行, 无论 taskA 和 taskB 是否将要运行:

```
taskB.mustRunAfter(taskA)
```

"should run after" 规则其实和 "must run after" 很像, 只是没有那么的严格, 在 2 种情况下它会被忽略:

1. 使用规则来阐述一个执行的循环.
2. 当并行执行并且一个任务的所有依赖除了 “should run after” 任务其余都满足了, 那么这个任务无论它的 “should run after” 依赖是否执行, 它都可以执行. (编者: 翻译待商榷, 提供具体例子)

总之, 当要求不是那么严格时, “should run after” 是非常有用的.

即使有目前的这些规则, 我们仍可以执行 taskA 而不管 taskB, 反之亦然.

例子 **15.14.** 加入 **'must run after'**

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.mustRunAfter taskX
```

gradle -q taskY taskX 的输出

```
> gradle -q taskY taskX
taskX
taskY
```

例子 **15.15.** 加入 **'should run after'**

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.shouldRunAfter taskX
```

gradle -q taskY taskX 的输出

```
> gradle -q taskY taskX
taskX
taskY
```

在上面的例子里, 我们仍可以直接执行 taskY 而不去 taskX :

例子 **15.16.** 任务排序不影响任务执行

gradle -q taskY 的输出

```
> gradle -q taskY
taskY
```

为了在 2 个任务间定义 “must run after” 或者 “should run after” 排序, 我们需要使用 Task.mustRunAfter() 和 Task.shouldRunAfter() 方法. 这些方法接收一个任务的实例， 任务的名字或者任何 Task.dependsOn()可以接收的输入.

注意 “B.mustRunAfter(A)” 或者 “B.shouldRunAfter(A)” 并不影响任何任务间的执行依赖:

- tasks A 和 B 可以被独立的执行. 排序规则只有当 2 个任务同时执行时才会被应用.
- 在运行时加上 --continue, 当 A 失败时 B 仍然会执行.

之前提到过, “should run after” 规则在一个执行循环中将被忽略:

例子 **15.17. 'should run after'** 任务的忽略

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
task taskZ << {
    println 'taskZ'
}
taskX.dependsOn taskY
taskY.dependsOn taskZ
taskZ.shouldRunAfter taskX
```

gradle -q taskX 的输出

```
> gradle -q taskX
taskZ
taskY
taskX
```

给 task 加入描述

你可以给你的任务加入一段描述性的文字. 它将会在任务执行的时候显示出来.

例子 **15.18.** 给任务加入描述

build.gradle

```
task copy(type: Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

替换 tasks

有时候你想要替换一个任务. 举个例子, 如果你想要互换一个通过 java 插件定义的任务和一个自定义的不同类型的任务:

例子 14.19. 覆写一个任务

build.gradle

```
task copy(type: Copy)

task copy(overwrite: true) << {
    println('I am the new one.')
}
```

gradle -q copy 的输出

```
> gradle -q copy
I am the new one.
```

这种方式将用你自己定义的任务替换一个 Copy 类型的任务, 因为它使用了同样的名字. 但你定义一个新的任务时, 你必须设置 overwrite 属性为 true. 否则的话 Gradle 会抛出一个异常, task with that name already exists.

跳过 tasks

Gradle 提供了好几种跳过一个任务的方式.

1. 使用判断条件 (predicate)

你可以使用 onlyIf() 方法来为一个任务加入判断条件. 就和 Java 里的 if 语句一样, 任务只有在条件判断为真时才会执行. 你通过一个闭包来实现判断条件. 闭包像变量一样传递任务, 如果任务应该被执行则返回真, 反之亦然. 判断条件在任务执行之前进行判断.

例子 **15.20.** 使用判断条件跳过一个任务

build.gradle

```
task hello << {
    println 'hello world'
}

hello.onlyIf { !project.hasProperty('skipHello') }
```

gradle hello -PskipHello 的输出

```
> gradle hello -PskipHello
:hello SKIPPED
BUILD SUCCESSFUL

Total time: 1 secs
```

2. 使用 StopExecutionException

如果想要跳过一个任务的逻辑并不能被判断条件通过表达式表达出来, 你可以使用 StopExecutionException. 如果这个异常是被一个任务要执行的动作抛出的, 这个动作之后的执行以及所有紧跟它的动作都会被跳过. 构建将会继续执行下一个任务.

例子 **15.21.** 通过 **StopExecutionException** 跳过任务

build.gradle

```
task compile << {
    println 'We are doing the compile.'
}

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw new StopExecutionException() }
}
task myTask(dependsOn: 'compile') << {
   println 'I am not affected'
}
```

gradle -q myTask 的输出

```
> gradle -q myTask
I am not affected
```

如果你直接使用 Gradle 提供的任务, 这项功能还是十分有用的. 它允许你为内建的任务加入条件来控制执行. [6]

3. 激活和注销 tasks

每一个任务都有一个已经激活的标记(enabled flag), 这个标记一般默认为真. 将它设置为假, 那它的任何动作都不会被执行.

例子 **15.22.** 激活和注销 **tasks**

build.gradle

```
task disableMe << {
    println 'This should not be printed if the task is disabled.'
}
disableMe.enabled = false
```

gradle disableMe 的输出

```
> gradle disableMe
:disableMe SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

跳过 up-to-date 的任务

如果你正在使用一些附加的任务, 比如通过 Java 插件加入的任务, 你可能会注意到 Gradle 会跳过一些任务, 这些任务后面会标注 **up-to-date**. 代表这个任务已经运行过了或者说是最新的状态, 不再需要产生一次相同的输出. 不仅仅是这些内建任务, 其实你在运行自己的任务时, 也会碰到这种情况.

1. 声明一个任务的输入和输出

让我们先看一个例子. 这里我们的任务会根据一个 XML 文件生成好几个输出文件. 让我们运行这个任务 2 次.

例子 **15.23. A generator task**

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

gradle transform 的输出

```
> gradle transform
:transform
Transforming source file.
```

gradle transform 的输出

```
> gradle transform
:transform
Transforming source file.
```

这里 Gradle 执行了这个任务两次, 即使什么都没有改变, 它也没有跳过这个任务. 这个例子里的任务, 它的行为是通过闭包定义的. Gradle 并不知道闭包会做什么, 也并不能自动指出是否这个任务是 up-to-date. 为了使用 Gradle 的 up-to-date 检测, 你需要定义任务的输入和输出.

每个任务都有输入和输出属性, 你需要使用这些属性来声明任务的输入和输出. 下面的例子中, 我们将声明 XML 文件作为输入, 并且把输出放在一个指定的目录. 让我们运行这个任务 2 次.

例子 **15.24.** 声明任务的输入和输出

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    inputs.file srcFile
    outputs.dir destDir
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

gradle transform 的输出

```
> gradle transform
:transform
Transforming source file.
```

gradle transform 的输出

```
> gradle transform
:transform UP-TO-DATE
```

现在, Gradle 就能够检测出任务是否是 up-to-date 状态.

任务的输入属性是 TaskInputs 类型. 任务的输出属性是 TaskOutputs 类型.

一个任务如果没有定义输出的话, 那么它永远都没用办法判断 up-to-date. 对于某些场景, 比如一个任务的输出不是文件, 或者更复杂的场景, TaskOutputs.upToDateWhen()) 方法会计算任务的输出是否应被视为最新.

总而言之, 如果一个任务只定义了输出, 如果输出不变的话, 它就会被视为 **up-to-date.**

2. 它是如何工作的?

当一个任务是首次执行时, Gradle 会取一个输入的快照 (snapshot). 该快照包含组输入文件和每个文件的内容的散列. 然后当 Gradle 执行任务时, 如果任务成功完成，Gradle 会获得一个输出的快照. 该快照包含输出文件和每个文件的内容的散列. Gradle 会保留这两个快照用来在该任务的下一次执行时进行判断.

之后, 每次在任务执行之前, Gradle 都会为输入和输出取一个新的快照, 如果这个快照和之前的快照一样, Gradle 就会假定这个任务已经是最新的 (up-to-date) 并且跳过任务, 反之亦然.

需要注意的是, 如果一个任务有指定的输出目录, 自从该任务上次执行以来被加入到该目录的任务文件都会被忽略, 并且不会引起任务过时 (out of date). 这是因为不相关任务也许会共用同一个输出目录. 如果这并不是你所想要的情况, 可以考虑使用 TaskOutputs.upToDateWhen())

Task 规则

有时候也想要一个任务的行为是基于已经定义好的取值范围或者特定规则, 下面的例子就提供了一种很直观漂亮的方式:

例子 **15.25.** 任务规则

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}
```

gradle -q pingServer1 的输出

```
> gradle -q pingServer1
Pinging: Server1
```

这里的 String 参数就是用来定义规则的.

规则并不只是在通过命令行使用任务的时候执行. 你也可以基于规则来创建依赖关系:

例子 **15.26.** 基于规则的任务依赖

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}

task groupPing {
    dependsOn pingServer1, pingServer2
}
```

gradle -q groupPing 的输出

```
> gradle -q groupPing
Pinging: Server1
Pinging: Server2
```

如果你运行 “gradle -q tasks”, 你并不能找到名叫 “pingServer1” 或者 “pingServer2” 的任务, 但是这个脚本仍然会执行这些任务.

终止 tasks

> 终止任务是一个正在开发的功能.

这里的终止任务并不是指终止一个任务, 而是指一个无论运行结果如何最后都会被执行的任务.

例子 **15.27.** 加入一个任务终止器

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}

taskX.finalizedBy taskY
```

gradle -q taskX 的输出

```
> gradle -q taskX
taskX
taskY
```

即使要终止的任务失败了, 终止任务仍会继续执行.

例子 **14.28.** 当任务失败时**k**

build.gradle

```
task taskX << {
    println 'taskX'
    throw new RuntimeException()
}
task taskY << {
    println 'taskY'
}

taskX.finalizedBy taskY
```

gradle -q taskX 的输出

```
> gradle -q taskX
taskX
taskY
```

另外, 如果要终止的任务并没有被执行 (比如上一节讲的 up-to-date) 那么终止任务并不会执行.

当构建创建了一个资源, 无论构建失败或成功时这个资源必须被清除的时候, 终止任务就非常有用.

要使用终止任务, 你必须使用 Task.finalizedBy()) 方法. 一个任务的实例, 任务的名称, 或者任何 Task.dependsOn() 可以接收的输入都可以作为这个任务的输入.

补充

下面补充的部分原本是第 **14** 章，最新的 **Gradle** 文档将其移除，所以将其作为补充放到这一章节。

Gradle 属性 和 system 属性

Gradle 提供了多种的方法让您可以在构建脚本中添加属性. 使用 -D 命令选项，您可以向运行 Gradle 的 JVM 传递一个 system 属性 . **Gradle** 命令的 -D 选项 和 **Java** 命令的 -D 选项有些相同的效果.

您也可以使用属性文件向您的 Project 对象中添加属性. 您可以在 Gradle 用户目录(如果您没有在 *USER_HOME*/.gradle 配置默认设置,则由"GRADLE_USER_HOME" 环境变量定义) 或者项目目录放置一个 gradle.properties 文件.如果是多项目的话，您可以在每个子目录里都放置一个 gradle.properties 文件. gradle.properties 文件内容里的属性能够被 Project 对象访问到. 不过有一点，用户目录中的 gradle.properties 文件优先权大于项目目录中的 gradle.properties 文件.

您也可以通过 -P 命令选项直接向Project 对象中添加属性.

另外，当 Gradle 看到特别命名的 system 属性或者环境变量时，Gradle 也可以设置项目属性. 比如当您没有管理员权限去持续整合服务，还有您需要设置属性值但是不容易时，这个特性非常有用. 出于安全的原因，在这种情况下，您没法使用 -P 命令选项，您也不能修改系统级别的文件. 确切的策略是改变您持续继承构建工作的配置，增加一个环境变量设置令它匹配一个期望的模式. 对于当前系统来说，这种方法对于普通用户来说是不可见的. [6]

如果环境变量的名字是 ORG_GRADLE_PROJECT=somevalue, Gradle 会使用值为 somevalue 在您的 Project 对象中设定一个支持属性. 另外 Gradle 也支持 system 属性，但是使用不同的名字模式，例如 org.gradle.project.prop .

您也可以在 gradle.properties 文件中设置 system 属性.如果一个属性名的前缀为 “systemProp”，那么这个属性和它的属性值会被设置为 system 属性. 如果没有这个前缀，在多项目构建中，除了根项目会被忽略外，“systemProp.” 属性会在任何项目中设置.也就是说仅仅根项目的 gradle.properties 文件会被检查其属性的前缀是否是 “systemProp”.

**例子 14.2.通过 gradle.properties 文件设置属性

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
sysProp=shouldBeOverWrittenBySysProp
envProjectProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

build.gradle

```
task printProps << {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
    println System.properties['system']

}
```

[6]. *Jenkins*, *Teamcity*, or *Bamboo* 都是 提供这个功能的 CI 服务.

使用 **gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=systemPropertyValue printProps** 输出

```
> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue    -Dorg.gradle.proje
ct.systemProjectProp=systemPropertyValue printProps
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
envPropertyValue
systemValue
```

使用其他的脚本配置项目

您还可以使用其他的构建脚本来配置当前的项目，Gradle 构建语言的所有的内容对于其他的脚本都是可以使用的. 您甚至可以在别的脚本中再使用其他的脚本.

例子 **14.3.**使用其他的构建脚本配置项目

build.gradle

```
apply from: 'other.gradle'
```

other.gradle

```
println "configuring $project"
task hello << {
    println' 'hello form other srcipt'
}
```

使用 **gradle -q hello** 输出

```
> gradle -q hello
configuring root project 'configureProjectUsingScript'
hello from other script
```

使用其他的脚本配置任意对象

您也可以使用其他的构建脚本配置任意的对象.

例子**: 14.5.**使用别的脚本配置配置对象

build.gradle

```
task config << {
   def pos = new java.text.FieldPosition(10)

   // 使用另一个脚本
   apply from: 'other.gradle', to: pos
   println pos.beginIndex
   println pos.endIndex

}
```

other.gradle

```
beginIndex = 1
endIndex = 5
```

使用 **gradle -q configure** 输出

```
> gradle -q configure
1
5
```

配置任意对象

您可以使用下面方法配置任意的对象.

例子 **14.4.**配置任意对象

build.gradle

```
task configure << {
   def pos = configure(new java.text.FieldPosition(10)) {
       beginIndex = 1
       endIndex = 5
   }

   println pos.beginIndex
   println pos.endIndex

}
```

使用 **gradle -q configure** 输出

```
> gradle -q configure
1
5
```

缓存

为了提高响应能力，Gradle 默认缓存了所有编译后的脚本. 包括所有的构建脚本，初始化脚本，还有其他脚本. Gradle 创建了一个 .gradle 目录来存放编译后的脚本，下次您运行构建脚本时，如果这个脚本自从它被编译后就再也没有被改动过，Gradle 会先使用编译后的脚本. 否则 Gradle 会重新编译脚本，然后将新编译后的文件缓存起来. 如果您使用 Gradle --recompile--scripts 运行脚本，缓存的脚本就会被删除，然后新编译后的文件就会再被缓存. 这种方法可以强制 Gradle 重新编译脚本并缓存.

文件操作

大多数构建工作需要操作文件，Gradle 增加了一些API帮助您处理这些工作。

Locating files

使用 `Project.file()` 方法能够相对项目目录定位一个文件

例 **16.1.** 定位文件

build.gradle

```
// 使用一个相对路径

File configFile = file('src/config.xml')

// 使用一个绝对路径

configFile = file(configFile.absolutePath)

// 使用一个项目路径的文件对象

configFile = file(new File('src/config.xml'))`
```

`file()` 方法接收任何形式的对象参数.它会将参数值转换为一个绝对文件对象,一般情况下，你可以传递一个 `String` 或者一个 `File` 实例.如果传递的路径是个绝对路径,它会被直接构造为一个文件实例.否则,会被构造为项目目录加上传递的目录的文件对象.另外, `file()函数` 也能识别 `URL` ,例如 file:/some/path.xml.

这个方法非常有用,它将参数值转换为一个绝对路径文件.所以请尽量使用 `new File(somePath)` , 因为 `file()` 总是相对于当前项目路径计算传递的路径,然后加以矫正.因为当前工作区间目录依赖于用户以何种方式运行 Gradle.

文件集合

文件集合表示一组文件，Gradle 使用 `FileCollection` 接口表示文件集合, Gradle API 中的许多项目都实现了这个接口,例如 dependency configurations .

获取 `FileCollection` 实例的一种方法是使用 `Project.files()` 方法.你可以传递任何数量的对象参数,这个方法能将你传递的对象集合转换为一组文件对象. `files()` 方法接收任何类型对象参数.每一个 `file()` 方法都依赖于项目目录(在第 15 章,第一小节中介绍). `files()` 方法也接收 `collections` , `iterables` , `maps` 和 `arrays` 类型参数.这些参数的内容会被解析，然后被转换为文件对象.

例 **15.2** 创建文件集合

build.gradle

```
FileCollection collection = files('src/file1.txt',
                                  new File('src/file2.txt'),
                                  ['src/file3.txt', 'src/file4.txt'])
```

文件集合可以被迭代器,使用迭代操作能够将其转换为其他的一些类型.你可以使用 `+` 操作将两个文件集合合并,使用 `-` 操作能够对一个文件集合做减法.下面一些例子介绍如何操作文件集合.

例 **15.3** 使用文件集合

build.gradle

```
// 对文件集合进行迭代
collection.each {File file ->
    println file.name
}

// 转换文件集合为其他类型
Set set = collection.files
Set set2 = collection as Set
List list = collection as List
String path = collection.asPath
File file = collection.singleFile
File file2 = collection as File

// 增加和减少文件集合
def union = collection + files('src/file3.txt')
def different = collection - files('src/file3.txt')
```

你也可以向 `files()` 方法专递一个闭合或者可回调的实例参数.当查询集合的内容时就会调用它,然后将返回值转换为一些文件实例.返回值可以是 `files()` 方法支持的任何类型的对象.下面有个简单的例子来演示实现 `FileCollection` 接口

例 **15.4** 实现一个文件集合

build.gradle

```
task list << {
    File srcDir

    // 使用闭合创建一个文件集合
    collection = files { srcDir.listFiles() }

    srcDir = file('src')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }

    srcDir = file('src2')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }
}
```

使用 `gradle -q list` 输出结果

```
> gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

另外, `files()` 方法也接收其他类型的参数:

FileCollection

内容损坏的文件包含在文件集合中.

Task

任务的输出文件包含在文件集合中.

TaskOutputs

`TaskOutputs` 的输出文件包含在文件集合中

值得注意的是当有需要时文件集合的内容会被被惰性处理,就比如一些任务在需要的时候会创建一个 `FileCollecion` 代表的文件集合.

文件树

文件树就是一个按照层次结构分布的文件集合,例如,一个文件树可以代表一个目录树结构或者一个 ZIP 压缩文件的内容.它被抽象为 `FileTree` 结构, `FileTree` 继承自 `FileCollection` ,所以你可以像处理文件集合一样处理文件树, Gradle 有些对象实现了 `FileTree` 接口,例如 源集合. 使用 `Project.fileTree()` 方法可以得到 `FileTree` 的实例,它会创建一个基于基准目录的对象,然后视需要使用一些 `Ant-style` 的包含和去除规则.

例 **15.5** 创建文件树 **build.gradle**

```
/以一个基准目录创建一个文件树
FileTree tree = fileTree(dir: 'src/main')

// 添加包含和排除规则
tree.include '**/*.java'
tree.exclude '**/Abstract*'

// 使用路径创建一个树
tree = fileTree('src').include('**/*.java')

// 使用闭合创建一个数
tree = fileTree('src') {
    include '**/*.java'
}

// 使用map创建一个树
tree = fileTree(dir: 'src', include: '**/*.java')
tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])
tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test*/**')
```

就像使用文件集合一样,你可以访问文件树的内容,使用 `Ant-style` 规则选择一个子树。

例 **15.6** 使用文件树 **build.gradle**

```
// 遍历文件树
tree.each {File file ->
    println file
}

// 过滤文件树
FileTree filtered = tree.matching {
    include 'org/gradle/api/**'
}

// 合并文件树A
FileTree sum = tree + fileTree(dir: 'src/test')

// 访问文件数的元素
tree.visit {element ->
    println "$element.relativePath => $element.file"
}
```

使用一个归档文件的内容作为文件树

你可以使用 ZIP 或者 TAR 等压缩文件的内容作为文件树, `Project.zipTree()` 和 `Project.tarTree()` 方法返回一个 `FileTree` 实例, 你可以像使用其他文件树或者文件集合一样使用它.例如,你可以使用它去扩展一个压缩文档或者合并一些压缩文档.

例 **15.7** 使用压缩文档作为文件树 **build.gradle**

```
// 使用路径创建一个 ZIP 文件
FileTree zip = zipTree('someFile.zip')

// 使用路径创建一个 TAR 文件
FileTree tar = tarTree('someFile.tar')

//tar tree 能够根据文件扩展名得到压缩方式,如果你想明确的指定压缩方式,你可以使用下面方法
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

指定一组输入文件

在 `Gradle` 中有一些对象的某些属性可以接收一组输入文件.例如, `JavaComplile` 任务有一个 `source` 属性,它定义了编译的源文件,你可以设置这个属性的值,只要 `files()` 方法支持. 这意味着你可以使用 `File` , `String` , `collection` , `FileCollection` 甚至是使用一个闭合去设置属性的值.

例 **15.8** 指定文件

build.gradle

```
//使用一个 File 对象设置源目录
compile {
    source = file('src/main/java')
}

//使用一个字符路径设置源目录
compile {
    source = 'src/main/java'
}

// 使用一个集合设置多个源目录
compile {
    source = ['src/main/java', '../shared/java']
}

// 使用 FileCollection 或者 FileTree 设置源目录
compile {
    source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/**' }
}

// 使用一个闭合设置源目录
compile {
    source = {
        // Use the contents of each zip file in the src dir
        file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect { zipTree(i
t) }
    }
}
```

Usually, there is a method with the same name as the property, which appends to the set of files. Again, this method accepts any of the types supported by the files() method.

通常情况下,会有一个方法名和属性名相同的方法能够附加一组文件,这个方法接收 `files()` 方法支持的任何类型的值.

例 **15.9** 指定文件

build.gradle

```
compile {
    // 使用字符路径添加源目录
    source 'src/main/java', 'src/main/groovy'

    // 使用 File 对象添加源目录
    source file('../shared/java')

    // 使用闭合添加源目录
    source { file('src/test/').listFiles() }
}
```

复制文件

你可以使用复制任务(`Copy`)去复制文件. 复制任务扩展性很强,能够过滤复制文件的内容, 映射文件名.

使用复制任务时需要提供想要复制的源文件和一个目标目录,如果你要指定文件被复制时的转换方式，可以使用 复制规则. 复制规则被 `CopySpec` 接口抽象,复制任务实现了这个接口. 使用 `CopySpec.from()` 方法指定源文件.使用 `CopySpec.into()` 方法指定目标目录.

例 **15.10.** 使用复制任务复制文件

build.gradle

```
task copyTask(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
}
```

`from()` 方法接收任何 `files()` 方法支持的参数. 当参数被解析为一个目录时,在这个目录下的任何文件都会被递归地复制到目标目录(但不是目录本身).当一个参数解析为一个文件时,该文件被复制到目标目录中.当参数被解析为一个不存在的文件时,这个参数就会忽略.如果这个参数是一个任务,任务的输出文件(这个任务创建的文件)会被复制,然后这个任务会被自动添加为复制任务的依赖.

例 **15.11** 指定复制任务的源文件和目标目录

build.gradle

```
task anotherCopyTask(type: Copy) {
    // 复制 src/main/webapp 目录下的所有文件
    from 'src/main/webapp'
    // 复制一个单独文件
    from 'src/staging/index.html'
    // 复制一个任务输出的文件
    from copyTask
    // 显式使用任务的 outputs 属性复制任务的输出文件
    from copyTaskWithPatterns.outputs
    // 复制一个 ZIP 压缩文件的内容
    from zipTree('src/main/assets.zip')
    // 最后指定目标目录
    into { getDestDir() }
}
```

你可以使用 `Ant-style` 规则或者一个闭合选择要复制的文件.

例 **15.12** 选择要复制文件

build.gradle

```
task copyTaskWithPatterns(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    include '**/*.html'
    include '**/*.jsp'
    exclude { details -> details.file.name.endsWith('.html') &&
                         details.file.text.contains('staging') }
}
```

你也可以使用 `Project.copy()` 方法复制文件,它的工作方式有一些限制,首先该方法不是增量的,请参考 第 14.9节 跳过最新的任务.第二,当一个任务被用作复制源时(例如 from() 方法的参数), `copy()` 方法不能够实现任务依赖,因为它是一个普通的方法不是一个任务.因此,如果你使用 `copy()` 方法作为一个任务的一部分功能,你需要显式的声明所有的输入和输出以确保获得正确的结果.

例 **15.13** 不使用最新检查方式下用 **copy()** 方法复制文件

build.gradle

```
task copyMethod << {
    copy {
        from 'src/main/webapp'
        into 'build/explodedWar'
        include '**/*.html'
        include '**/*.jsp'
    }
}
```

例 **15.14** 使用最新的检查方式下用 **copy()** 方法复制文件

build.gradle

```
task copyMethodWithExplicitDependencies{

     // 对输入做最新检查,添加 copyTask 作为依赖
    inputs.file copyTask
    outputs.dir 'some-dir' //对输出做最新检查
    doLast{
        copy {
            // 复制 copyTask 的输出
            from copyTask
            into 'some-dir'
        }
    }
}
```

建议尽可能的使用复制任务,因为它支持增量化的构建和任务依赖推理，而不需要去额外的费力处理这些.不过 `copy()` 方法可以用作复制任务实现的一部分.即该 方法被在自定义复制任务中使用,请参考 第60章 编写自定义任务.在这样的场景下，自定义任务应该充分声明与复制操作相关的输入/输出。

15.6.1 重命名文件

例 **15.15** 在复制时重命名文件

build.gradle

```
task rename(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // 使用一个闭合映射文件名
    rename { String fileName ->
        fileName.replace('-staging-', '')
    }
    // 使用正则表达式映射文件名
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(.+)/, '$1$2')
}
```

15.6.2 过滤文件

例 **15.16** 在复制时过滤文件

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

task filter(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // 在文件中替代属性标记
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // 使用 Ant 提供的过滤器
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // 用一个闭合来过滤每一行
    filter { String line ->
        "[$line]"
    }
    // 使用闭合来删除行
    filter { String line ->
        line.startsWith('-') ? null : line
    }
}
```

在源文件中扩展和过滤操作都会查找的某个标志 `token`,如果它的名字是 `tokenName` , 它的格式应该类似于 `@tokenName@` .

15.6.3 使用 CopySpec 类

复制规范来自于层次结构,一个复制规范继承其目标路径,包括模式,排除模式，复制操作,名称映射和过滤器.

例 **15.17.** 嵌套复制规范

build.gradle

```
task nestedSpecs(type: Copy) {
    into 'build/explodedWar'
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html'
    }
    into('libs') {
        from configurations.runtime
    }
}
```

使用同步任务

同步任务(`Sync`)任务继承自复制任务(`Copy`),当它执行时,它会复制源文件到目标目录中,然后从目标目录中的删除所有非复制的文件,这种方式非常有用,比如安装一个应用,创建一个文档的副本,或者维护项目的依赖关系副本.

下面有一个例子,维护 `build/libs` 目录下项目在运行时的依赖

例 **15.7** 使用 **Sync** 任务复制依赖关系

build.gradle

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

创建归档文件

一个项目可以有很多 JAR 文件,你可以向项目中添加 WAR , ZIP 和 TAR 文档,使用归档任务可以创建这些文档: Zip , Tar , Jar , War 和Ear. 它门都以同样的机制工作.

例 **15.19** 创建一个 **ZIP** 文档

build.gradle

```
apply plugin: 'java'

task zip(type: Zip) {
    from 'src/dist'
    into('libs') {
        from configurations.runtime
    }
}
```

所有的归档任务的工作机制和复制任务相同,它们都实现了 `CopySpec` 接口,和 `Copy` 任务一样,使用 `from()` 方法指定输入文件,可以选择性的使用 `into()` 方法指定什么时候结束.你还可以过滤文件内容,重命名文件等等,就如同使用复制规则一样.

重命名文档

`projectName-vsersion.type` 格式可以被用来生成文档名,例如:

例 **15.20** 创建压缩文档

build.gradle

```
apply plugin: 'java'

version = 1.0

task myZip(type: Zip) {
    from 'somedir'
}

println myZip.archiveName
println relativePath(myZip.destinationDir)
println relativePath(myZip.archivePath)
```

使用 `gradle -q myZip` 命令进行输出:

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/zipProject-1.0.zip
```

上面例子使用一个名为 myZip的 Zip 归档任务生成名称为 zipProject-1.0.zip 的 ZIP 文档,区分文档任务名和最终生成的文档名非常的重要

可以通过设置项目属性 `archivesBaseName` 的值来 修改生成文档的默认名.当然,文档的名称也可以通过其他方法随时更改.

归档任务中有些属性可以配置,如表 15.1 所示:

表 **15.1** 归档任务-属性名

属性名	类型	默认值	描述
archiveName	String	baseName-appendix-version-classifier.extension,如果其中任何一个都是空的,则不添加名称	归档文件的基本文件名
archivePath	File	destinationDir/archiveName	生成归档文件的绝对路径。
destinationDir	File	取决于文档类型, JAR 和 WAR 使用 project.buildDir/distributions. project.buildDir/libraries.ZIP 和 TAR	归档文件的目录
baseName	String	project.name	归档文件名的基础部分。
appendix	String	null	归档文件名的附加部分。
version	String	project.version	归档文件名的版本部分。
classifier	String	null	归档文件名的分类部分
extension	String	取决于文档类型和压缩类型: zip, jar, war, tar, tgz 或者 tbz2.	归档文件的扩展名

例 **15.21** 配置归档文件-自定义文档名

build.gradle

```
apply plugin: 'java'
version = 1.0

task myZip(type: Zip) {
    from 'somedir'
    baseName = 'customName'
}

println myZip.archiveName
```

使用 `gradle -q myZip` 命令进行输出:

```
> gradle -q myZip
customName-1.0.zip
```

更多配置:

例 **15.22** 配置归档任务- 附加其他后缀

build.gradle

```
apply plugin: 'java'
archivesBaseName = 'gradle'
version = 1.0

task myZip(type: Zip) {
    appendix = 'wrapper'
    classifier = 'src'
    from 'somedir'
}

println myZip.archiveName
```

使用 `gradle -q myZip` 命令进行输出:

```
> gradle -q myZip
gradle-wrapper-1.0-src.zip
```

多个文档共享内容

你可以使用 `Project.copySpec()` 在不用归档文件间共享内容. 如果你想发布一个文档,然后在另一个项目中使用，这部分将在第 53 章 发布文档 中讲述.

在 Gradle中使用Ant

Gradle 出色的集成了 Ant. 你可以在 Gradle 构建时使用单独的 Ant 任务或完整的 Ant 构建. 事实上, 你会发现在 Gradle 构建脚本中使用Ant任务远比直接使用 Ant 的 XML 格式更加容易和强大. 你甚至可以将 Gradle 仅仅作为一个强大的 Ant 脚本工具.

Ant 可以分为两层. 第一层是 Ant 语言. 它给 build.xml 文件, 处理目标, 像 macrodefs 的特殊构造等提供语法支持. 换句话说, 除了 Ant 任务和类型, 其余一切都支持. Gradle 了解这种语言, 并允许用户直接导入 Ant 的 build.xml 到 Gradle 的项目下. 然后, 你可以 像 Gradle 任务一样直接使用这些 Ant 构建.

第二层是 Ant 丰富的任务和类型, 如 javac, copy 或 jar. Gradle 提供了基于 Groovy 的集成以及梦幻般的 AntBuilder.

最后，由于构建脚本是 Groovy 脚本, 你总是可以执行一个 Ant 构建作为一个外部进程. 构建脚本可能会含有类似的语句："ant clean compile".execute().[7]

你可以使用 Gradle 的 Ant 集成，作为迁移路径将构建 Ant 迁移到 Gradle. 例如, 你可以从通过导入现有的 Ant 构建开始, 然后将你的依赖声明从 Ant 脚本迁移到你的 build 文件. 最后, 你可以将任务迁移到你的构建文件, 或者用一些 Gradle 插件代替他们. 随着时间的推移, 这部分工作会一步一步地完成, 并且你可以在整个进程中有一个运行良好的 Gradle 构建.

16.1.使用 Ant 任务和 Ant 类型的构建

在构建脚本中, Ant 属性是由 Gradle提供的. 这是一个用于参考的 AntBuilder 实例. AntBuilder 用于从构建脚本访问 Ant 任务, 类型和属性. 下面的例子解释了从 Ant 的 `build.xml` 格式到 Grooy 的映射.

你可以通过调用 AntBuilder 实例的方法执行 Ant 任务. 你可以使用任务名称作为方法名, 比如, 可以通过调用 `anto.echo()` 任务执行 Ant `echo` 任务. Ant 任务属性通过 Map 参数传递给方法. 下面是一个 `echo` 任务的例子, 注意我们也可以混合使用 Groovy 代码和 Ant 任务标记, 这点个功能非常强大.

例 **16.1.**使用 **Ant** 任务

build.gradle

```
task hello << {
    String greeting = 'hello from Ant'
    ant.echo(message: greeting)
}
```

`gradle hello` 的输出

```
>\> gradle hello
>:hello
>[ant:echo] hello from Ant
>
>BUILD SUCCESSFUL
>
>Total time: 1 secs
```

你可以添加嵌套元素添加到一个封闭的 Ant 任务的内部. 定义嵌套元素跟定义任务的方式相同, 通过与调用我们要定义的元素名相同的方法.

例 **16.3.**添加嵌套元素到一个**Ant**任务

build.gradle

```
task zip << {
    ant.zip(destfile: 'archive.zip') {
        fileset(dir: 'src') {
            include(name: '**.xml')
            exclude(name: '**.java')
        }
    }
}
```

你可以像访问任务一样访问 Ant type,只需要将 type 名作为方法名即可. 该方法调用返回的 Ant 数据类型，可以直接在构建脚本中使用.下面的例子中,我们创建了一个 `ant path` 对象,然后遍历他的内容.

例 **16.4.**使用 **Ant** 类型

build.gradle

```
task list << {
    def path = ant.path {
        fileset(dir: 'libs', includes: '*.jar')
    }
    path.list().each {
        println it
    }
}
```

更多关于 AntBuilder 的信息可以参考 'Groovy in Action'8.4 或者Groovy Wiki.

在构建中使用自定义Ant任务

为了让你的构建可以自定义任务, 你可以使用 `taskdef(通常更容易)` 或者 `typedef` Ant 任务, 就像你在一个 `build.xml` 文件中一样. 然后，你可以参考内置 Ant 任务去定制 Ant 任务.

例 **16.5.**使用自定义 **Ant** 任务

build.gradle

```
task check << {
    ant.taskdef(resource: 'checkstyletask.properties') {
        classpath {
            fileset(dir: 'libs', includes: '*.jar')
        }
    }
    ant.checkstyle(config: 'checkstyle.xml') {
        fileset(dir: 'src')
    }
}
```

你可以使用 Gradle 的依赖管理去组装你自定义任务所需要的 classpath. 要做到这一点, 你需要定义一个自定义配置类路径, 然后添加一些依赖配置.在Section 51.4, “How to declare your dependencies”部分有更详细的说明.

例 **16.6.**声明类路径的自定义 **Ant** 任务

build.gradle

```
configurations {
    pmd
}

dependencies {
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}
```

要使用 classpath 配置, 使用自定义配置的 `asPath` 属性。

例 **16.7.**一起使用自定义**Ant**任务和依赖管理

build.gradle

```
task check << {
    ant.taskdef(name: 'pmd',
                classname: 'net.sourceforge.pmd.ant.PMDTask',
                classpath: configurations.pmd.asPath)
    ant.pmd(shortFilenames: 'true',
            failonruleviolation: 'true',
            rulesetfiles: file('pmd-rules.xml').toURI().toString()) {
        formatter(type: 'text', toConsole: 'true')
        fileset(dir: 'src')
    }
}
```

导入一个Ant构建

你可以在你的gradle项目中通过 `ant.importBuild()` 来导入一个ant构建,当你导入了一个ant构建,每一个 `ant target` 都会被视为一个Gradle任务.这意味着你可以像操作,执行gradle任务一样操作,执行 `ant target`

例 **16.8.**导入**ant**构建

build.gradle

```
ant.importBuild 'build.xml'
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

`gradle hello` 的输出

```
>\> gradle hello
>:hello
>[ant:echo] Hello, from Ant
>
>BUILD SUCCESSFUL
>
>Total time: 1 secs
```

你可以添加一个依赖于 `ant target` 的任务: 例 **16.9.**依赖于**ant target**的任务

build.gradle

```
ant.importBuild 'build.xml'

task intro(dependsOn: hello) << {
    println 'Hello, from Gradle'
}
```

> `gradle intro` 的输出 > gradle intro :hello [ant:echo] Hello, from Ant :intro Hello, from Gradle
>
> BUILD SUCCESSFUL
>
> Total time: 1 secs

或者,你可以为 `ant target` 添加动作

例 **16.10.** 为 **Ant target** 添加动作

build.gradle

```
ant.importBuild 'build.xml'

hello << {
    println 'Hello, from Gradle'
}
```

`gradle hello` 的输出

> > gradle hello :hello [ant:echo] Hello, from Ant Hello, from Gradle
>
> BUILD SUCCESSFUL
>
> Total time: 1 secs

当然,一个 `ant target` 也可以依赖于gradle的任务

例 **16.11.** 为 **Ant target** 添加动作

build.gradle

```
ant.importBuild 'build.xml'

task intro << {
    println 'Hello, from Gradle'
}
```

build.xml

```
<project>
    <target name="hello" depends="intro">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

`gradle hello` 的输出

> gradle hello :intro Hello, from Gradle :hello [ant:echo] Hello, from Ant
>
> BUILD SUCCESSFUL
>
> Total time: 1 secs

有时候可能需要'重命名' `ant target` 以避免与现有的gradle任务冲突.需要使用AntBuilder.importBuilder())方法.

例 **16.12.** 重命名导入的**ant target**

build.gradle

```
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

`gradle a-hello` 的输出

> gradle a-hello :a-hello [ant:echo] Hello, from Ant
>
> BUILD SUCCESSFUL
>
> Total time: 1 secs

注意,方法的二个参数应该是一个TransFormer,在Groovy编程的时候，由于Groovy的支持自动闭包单抽象方法的类型。我们可以简单地使用闭包取代匿名内部类，

Ant的属性与引用

有许多方法可以设定 Ant 属性,可以通过Ant任务使用属性.您可以直接在 `AntBuilder` 的实例设置属性。Ant的属性也可以作为一个可改变的 `Map` .也可以使用Ant的任务属性,如下例所示:

例**16.13.**设置**Ant**属性

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
```

build.xml

```
<echo>buildDir = ${buildDir}</echo>
```

许多任务会在执行时设置属性.下面有几种方法来获取属性值,可以直接从 `AntBuilder` 实例获得属性,如下所示,ant的属性仍然是作为一个map:

例**16.14.**获取**Ant**属性

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
```

build.gradle

```
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

设置一个ant引用的方法:

例**16.15.**设置一个**Ant**引用

build.gradle

```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
```

build.xml

```
<path refid="classpath"/>
```

获取Ant引用的方法:

例**16.16.**获取一个**Ant**引用

build.xml

```
<path id="antPath" location="libs"/>
```

build.gradle

```
println ant.references.antPath
println ant.references['antPath']
```

API

Ant 集成是由 AntBuilder 提供的.git

Logging

Log 是构建的主要"UI"工具. 如果日志太过冗长, 那么真正的警告和问题会隐藏其中, 另一方面, 如果你出错了,你又需要搞清楚相关错误信息. Gradle 提供了6个等级的 log, 如表17.1.Logs Level所示.出了那些你可能经常看到的, 还有两个是 Gradle 特定级别的日志,被称为*QUIET*和*LIFECYCLE*.后者是默认的, 并用于报告生成进度.

表**17.1.Logs Level**

Level	用途
ERROR	错误信息
QUIET	重要消息信息
WARNING	警告信息
LIFECYCLE	进度消息信息
INFO	信息消息
DEBUG	调试信息

Choosing a log level

你可以在命令行中选择如表 17.2.Log 等级命令行选项所示的选项选择不同的日志级别.如表 17.3.堆栈信息选项中所示的选项来选择堆栈信息.

表**17.2.Log** 等级命令行选项

选项	输出日志等级
no logging options	LIFECYCLE及更高
-q or --quiet	QUIET及更高
-i or --info	INFO及更高
-d or --debug	DEBUG及更高(所有日志信息)

表 **17.3.**堆栈信息选项

选项	含义
No stacktrace options	无堆栈踪迹输出到控制台的情况下生成错误信息(如编译错误),仅在内部异常时打印堆栈信息.如果选择DEBUG日志等级,总会打印截断堆栈信息
-s or --stacktrace	打印截断堆栈信息,我们推荐这个而不是 `full stacktrace` ,Groovy的 `full stacktrace` 非常详细.(由于底层的动态调用机制。然而，他们通常不包含你的代码出了什么错的相关信息)
-S or --full-stacktrace	打印全部堆栈信息

编写自己的日志信息

用于记录在你的构建文件的简单方法是将消息写入标准输出.Gradle重定向任何东西写入到标准输出到它的log系统作为 `QUITE` 级别的log.

例 **17.1.**使用标准输出写入**log**信息

build.gradle

```
println 'A message which is logged at QUIET level'
```

摇篮还提供了一个 `logger` 属性来构建脚本，这是Logger的一个实例.这个接口继承自 `SLF4J` 接口并且加入了一F些Gradle的具体方法.下面是如何在构建脚本中使用此方法的例子：

例 **17.2.**写入自己的**log**信息

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.')
```

你还可以在构建中将其他类直接挂接到Gradle的log系统中(例如 `buildSrc` 目录下的类).只使用 `SLF4J logger` ,使用这个 `logger` 的方式与构建脚本提供的 `logger` 方式相同.

例 **17.3.**使用**SLF4J**写入**log**信息

build.gradle

```
import org.slf4j.Logger
import org.slf4j.LoggerFactory

Logger slf4jLogger = LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4j')
```

从外部工具和库记录日志

在内部, Gradle 使用 Ant 和 Ivy , 都有自己的 log 系统, Gradle 重定向他们的日志输出到 Gradle 日志系统. 除了Ant/Ivy的 `TRACE` 级别的日志, 映射到Gradle的 `DEBUG` 级别, 其余的都会有一个1:1的映射从 Ant/Ivy 的日志等级到 Gradle 的日志等级. 这意味着默认的 Gradle 日志级别将不会显示任何的 Ant /Ivy 的输出, 除非它是一个错误或警告.

有许多工具仍然使用标准输出记录,默认的,Gradle将标准输出重定向到 `QUIET` 的日志级别和标准错误的 `ERROR` 级别.该行为是可配置的.该项目对象提供了一个LoggerManager,当你构建脚本进行评估的时候,允许你改变标准输出或错误重定向的日志级别。

例 **17.4.**配置标准输出捕获

build.gradle

```
logging.captureStandardOutput LogLevel.INFO
println 'A message which is logged at INFO level'
```

任务同样提供了LoggingManager去更改任务执行过程中的标准输出或错误日志级别。

例 **17.5.**为任务配置标准输出捕获

build.gradle

```
task logInfo {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

Gradle同样提供了 `Java Util Logging` , `Jakarta Commons Logging` 和 `Log4j logging` 的集成工具.

使用这些工具包编写的构建的类的记录的任何日志消息都将被重定向到Gradle的日志记录系统。

改变Gradle的记录内容

你可以用自己的内容取代大部分摇篮的UI记录.你可以这样做，例如，如果你想以某种方式定制UI,如:记录更多或更少的信息，或更改log的格式.你可以使用Gradle.useLogger())方法替换日志.可从一个构建脚本或初始化脚本，或通过嵌入API替换.注意，这会完全禁用Gradle的默认输出.下面的初始化脚本改变任务执行和完成构建后日志的输出.

例 **17.6.**定制**Gradle logs**

init.gradle

```
useLogger(new CustomEventLogger())

class CustomEventLogger extends BuildAdapter implements TaskExecutionListener {

    public void beforeExecute(Task task) {
        println "[$task.name]"
    }

    public void afterExecute(Task task, TaskState state) {
        println()
    }

    public void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

`gradle -I init.gradle build` 的输出

```
> gradle -I init.gradle build [compile] compiling source

[testCompile] compiling test source

[test] running unit tests

[build]

build completed
```

你的 `logger` 可以实现下面列出的任何监听器接口.仅它实现接口被替换,其他接口保持不变。你可以在Section 56.6, “Responding to the lifecycle in the build script”中找到更多相关信息.

- BuildListener

- ProjectEvaluationListener
- TaskExecutionGraphListener
- TaskExecutionListener
- TaskActionListener

The Gradle Daemon

什么是 Gradle 的守护进程

维基百科中守护进程的解释

> 守护进程是一个运行后台进程, 非交互式用户直接控制的在计算机程序

Gradle 守护进程是一个后台进程, 它运行着繁重的构建, 然后在构建等待下一次构建的之间保持自身存在. 这使得数据和代码在下一次构建前已经准备好,并存入内存中. 这显著的提高了后续构建的性能. 启用Gradle守护进程是一种节约构建时间的廉价方式.

强烈建议在所有开发机器上启用*Gradle*的守护进程.但是不推荐在持续集成和构建服务器环境下启用守护进程(参见 Section 18.3, “When should I not use the Gradle Daemon?”).

Gradle自动管理守护进程.如果构建环境配置为利用后台程序,如果在没有可用守护进程,就会自动启动一个守护进程,或者使用现有的空闲的兼容守护进程.如果一个守护进程在3个小时内没有使用,它将会自我终结.一旦配置开发环境为使用的守护进程,守护进程通常是隐形的,容易被遗忘的.

管理和配置

如何启动Gradle的守护进程

在使用Gradle命令行接口时, `--daemon` 和 `--no-daemon` 命令行选项调用在单个构建时选择启用或禁用后台守护进程.通常,允许后台守护进程在一个环境中(例如一个用户账户)更为方便,可以使所有构建使用守护进程,而不需要记住 `--daemon` 开关.

有两种推荐的方式使守护进程持续与环境：

1. 通过环境变量 - 给 `GRADLE_OPTS` 环境变量添加 `-Dorg.gradle.daemon=true` 标识
2. 通过属性文件 - 给 `<<GRADLE_USER_HOME>>/gradle.properties` 文件添加 `org.gradle.daemon=true`

> 注意: `<<GRADLE_USER_HOME>>` 默认为 `<<USER_HOME>>/.gradle` , `<<USER_HOME>>为当前用户home目录` ,这个位置可以通过 `-g` 和 `-gradle-user-home` 命令行选项,以及由 `GRADLE_USER_HOME` 环境变量 `org.gradle.user.home` JVM系统属性配置。

这两种方法有同样的效果,使用哪一个是由个人喜好.大多数Gradle用户选择第二个方式,给gradle.properties并添加条目.

在Windows中，该命令将使当前用户启用守护：

```
 (if not exist "%HOMEPATH%/.gradle" mkdir "%HOMEPATH%/.gradle") && (echo foo >>
"%HOMEPATH%/.gradle/gradle.properties")
```

在类Unix操作系统，以下的Bash shell命令将使当前用户启用守护进程：

```
 touch ~/.gradle/gradle.properties && echo "org.gradle.daemon=true" >>
~/.gradle/gradle.properties
```

一旦以这种方式在构建环境中启用了守护进程,所有的构建将隐含一个守护进程.

如何禁用Gradle的守护进程

一般Gradle守护进程默认不启用.然而,一旦它被启用,有事希望对某些项目或某些构建禁用守护进程.

`--no-daemon` 命令行选项可用于强制守护进程不能用于该构建.这很少使用,但是在调试具有一定的构建或Gradle插件问题时,有时会很有用.在构建环境中,此命令行选项具有最高优先级.

怎样抑制“please consider using the Gradle Daemon”消息

Gradle可能会在构建结束时发出建议您使用Gradle守护进程的末尾警告.为了避免这个警告,您可以通过上述的这些方法使用守护进程,或者明确禁用守护进程.您可以通过上述的 `--no daemon` 的命令行选项明确禁用守护进程,或使用上述的 `org.gradle.deamon` 的值设置为 `false` 代替 `trie` .

因为不建议在持续集成构建中使用守护进程,如果 `CI` 环境变量已存在,Gradle不会发出此消息.

为什么会在机器上出现不只一个守护进程

有几个原因Gradle会创建一个新的守护进程代替使用一个已存在的守护进程.如果守护进程没有闲置,兼容,则会启动一个新的守护进程.

空闲的守护进程是当前未执行构建或做其他有用的工作.

兼容的守护进程是一个可以（或者可以达到）满足要求的编译环境的要求。Java安装程序运行的构建是构建环境方面的一个例子。构建运行时所需的JVM系统属性是另一个例子。

一个已经运行的Java进程可能不能满足所需的构建环境的某些方面。如果守护进程由Java7启动，但要求的环境要求为Java8,则守护进程是不兼容的，必须另外启动。再者，在运行的JVM不能改变一个运行时的某些性能。如内存分配（如-Xmx1024m），默认文本编码运行的JVM中，默认的语言环境，等等一个JVM不能改变的运行环境。

"Required build environment"通常在构建客户端(如Gradle命令行,IDE等)方面隐含构建环境,并明确通过命令行选项设置.参见Chapter 20,The Build Environment有关如何指定和控制构建环境的详细信息.

一下JVM系统属性是有效不变的.如果需求编译环境需要这些属性,不同的守护进程JVM在下列属性中有不同的值时,守护进程不兼容.

- file.encoding
- user.language
- user.country
- user.variant
- com.sun.management.jmxremote

下列JVM属性,通过启动参数控制,也是有效不变的.在需求构建环境和守护进程环境的对应属性必须按顺序完全匹配,才可兼容.

- 最大堆大小(即 -Xmx JVM参数)
- 最小堆大小(即 -Xms JVM参数)
- 引导类路径(即 -Xbootclasspath JVM参数)
- "assertion"状态(即 -ea 参数)

所需的Gradle版本是需求构建环境的另一个方面.守护进程被耦合到特定Gradle运行时,多个正在运行的守护进程产生的原因是使用使用不同版本的Gradle会在会话过程中处理多个项目.

守护进程占用多大内存并且能不能给它更大的内存?

如果需求构建环境没有指定最大堆内存,守护进程会使用多达1G的堆内存.它将会使用默认的JVM的最小堆内存.1G内存足够应付大多数构建.有数百个子项的构建,大量配置或者源码需求,或者要求有更好的表现,则需要更多地内存

为了提高守护进程可以使用的内存,指定相应的标志作为需求构建环境的一部分,请参见 Chapter 20. The Build Environment的详细信息.

如何停止守护进程

守护进程会在闲置3小时后自动终止.如果想在这之前停止守护进程,也可以通过操作系统运行 `gradle --stop` 命令终止后台进程. `--stop` 选项会要求所有运行相同版本的守护进程终止.

守护进程何时会出错

许多工程设计已经加入守护进程使得守护进程在日常的开发中变得更加健壮,透明和不起眼.无论如何,守护进程偶尔也会损坏或者枯竭.一个Gradle构建执行源自多个来源的任意代码.即使Gradle本身与守护进程的设计是经过严格的测试的,但是用户的构建脚本,或第三方插件可以通过诸如内存泄露,或腐化全局声明等缺陷来动摇守护进程.

另外,也可以通过构建时进行不正确的资源释放,也可能会动摇守护进程(构建环境正常).在在Microsoft Windows下是一个特别尖锐的问题,在程序读写文件后关闭失败的处理是非常随意的.

如果出现守护进程不稳定情况,可以简单的终止.回顾一下 `--no-daemon` 的选项可以用于构建阻止使用守护进程,这对于检验一个问题的罪魁祸首是不是守护进程很有帮助.

什么时候不使用Gradle守护进程

建议在开发环境中使用Gradle的守护进程,不建议在持续集成环境和构建服务器环境中使用守护进程.

守护进程可以更快的构建,这对于一个正坐在椅子前构建项目的人来说非常重要.对于CI构建来说,稳定性和可预见性是最重要的.为每个构建运行时用一个新的,完全孤立于以前的版本的程序,更加可靠。

工具和集成开发环境

Gradle工具API(参见Chapter.65.Embedding Gradle),用于IDEs和其他工具整合Gradle,总是使用Gradle守护进程执行构建.如果你是从IDE内部执行构建,那么你是在使用守护进程,而且不需要在你的环境中允许Gradle守护进程.

但是，除非您已明确启用的Gradle守护进程在你的环境的,你在命令行中的构建不会使用摇篮守护进程。

摇篮守护进程如何使构建速度更快

Gradle守护进程是一个常驻构建进程.在两个构建之间的空闲期间会等待着下次构建.与每个构建加载Gradle到内存相比,对于多个构建只需要加载一次Gradle到内存具有明显的好处.这本身就是对性能的显著优化,但是不止这些.

现代JVM的显著优化是运行时代码优化.例如,热点(HotSpot)(由Oracle提供并作为OpenJDK的基础的JVM实现)适用于优化运行时代码.优化是渐进的，而不是瞬间的。也就是说,代码在运行期间逐步优化,这意味着后续版本纯粹是基于这个优化过程变得更快.HotSpot实验表明,它需要5至10某处构建以优化至稳定.在一个守护进程的第一个构建和第十之间感知的编译时间的差异可以说是相当巨大的.

守护程序还允许更有效地在内存中缓存整个构建。例如，需要构建（如插件，构建脚本）的类可以在内存中举行的构建。同样，摇篮可保持在内存中缓存的构建数据的诸如的任务输入和输出的哈希值，用于增量构建。

未来可能的改进

目前，守护使构建速度更快有效地支持在内存中缓存和由JVM优化使代码更快。在未来的Gradle版本中，守护进程将变得更加聪明，预先完成工作。它可能，例如，在编辑完构建脚本后就开始下载依赖生成是将要运行的假设下后立即和新改变或添加的依赖性是必需的。

有许多方式使得在未来的版本的gradle的gradle守护进程。

Gradle 插件

Gradle 的核心为真实世界提供了很少的自动化. 所有的实用特性,类似编译java源码的能力, 是由插件提供的. 插件添加了新的任务(如:JavaCompile),域对象(如:SourceSet),公约(如:Java资源位置是`src/main/java`)以及来自其他插件延伸核心对象和对象。

在本章中，我们将讨论如何使用插件和关于插件的周边概念和术语。

插件的作用是什么

应用插件到项目允许插件来扩展项目的能力。它可以做的事情，如：

- 扩展摇篮模型（如:添加可配置新的DSL元素）
- 按照惯例配置项目(如:添加新的任务或配置合理的默认值)
- 应用特定的配置（如:增加组织库或执行标准）

通过应用插件,而不是向项目构建脚本添加逻辑,我们可以收获很多好处.应用插件:

- 促进重用和减少维护在多个项目类似的逻辑的开销
- 允许更高程度的模块化，提高综合性和组织
- 封装必要的逻辑，并允许构建脚本尽可能是声明性地

插件的类型

在Gradle中一般有两种类型的插件,脚本插件和二进制插件.脚本插件是额外的构建脚本,它会进一步配置构建,通常实行声明的方式操纵的构建.尽管他们可以外部化并且从远程位置访问,它们通常还是会在构建内部中使用.二进制插件是实现了Plugin接口的类,并且采用编程的方式来操纵构建.二进制插件可以驻留在构建脚本,项目层级内或外部的插件jar.

应用插件

插件需要声明被应用,通过Project.apply()方法完成.应用的插件是*idempotent*[注1],即相同的插件可以应用多次.如果插件先前以被应用,任何后来的应用是安全的,不会有任何影响的.

[1]译注:英文直接翻译的意思是幂等(denoting an element of a set that is unchanged in value when multiplied or otherwise operated on by itself.),上下中的大意应该是不会受其他因素的影响.

脚本插件

Example 21.1. Applying a script plugin

build.gradle

```
apply from: 'other.gradle'
```

脚本插件可以从本地文件系统或在远程位置的脚本中应用.文件系统的位置是相对于项目目录,而远程脚本位置的是由一个 `HTTP URL` 指定的.多个脚本插件（两种形式之一）可以被应用到给定的构建。

二进制插件

Example 21.2. Applying a binary plugin

build.gradle

```
apply plugin: 'java'
```

插件可以使用插件*ID*应用.插件的id作为给定的插件的唯一标识符.核心插件注册一个可以用作插件的id的短名称.在上述情况下,我们可以使用简称 `java` 的插件以应用JavaPlugin.社区插件,一方面会使用一个完全合格的形式的插件id(如 `com.github.foo.bar`),但还是有一些传统的插件可能仍然使用很短的，不合格的格式.

不使用一个插件的id，插件也可以通过简单地指定类来应用插件：

Example 21.3. Applying a binary plugin by type

build.gradle

```
apply plugin: JavaPlugin
```

在上面的例子中,JavaPlugin是指JavaPlugin,此类不是严格需要导入org.gradle.api.plugins包中的所有自动导入构建脚本（见:附录E，现有的IDE支持，以及如何没有它应付).此外，这是没有必要追加的.class以识别一个类常量在Groovy，因为它是在Java中。

二进制插件的位置

一个插件是一个简单的实现了插件接口的类.Gradle提供的核心插件作为其分布的一部分,因此,你需要做的仅仅是应用上述的插件.然而,非核心二进制插件需要到构建类路径才能应用它们.这可以以多种方式，包括以下方式实现:

- 定义插件作为构建脚本中内嵌类的声明.
- 定义插件为在项目中buildSrc目录下的一个源文件.(参见Section 62.4, “Build sources in the buildSrc project”)
- 包含来自外部jar定义的构建脚本依赖插件(参见Section 21.4, “Applying plugins with the buildscript block”)
- 包含插件DSL语言组成的插件门户网站(Section 21.5, “Applying plugins with the plugins DSL”)

欲了解有关自定义插件的跟多信息,参见Chapter 61, Writing Custom Plugins

使用构建脚本块应用插件

项目可以通过添加向构建脚本中加入插件的类路径然后在应用插件,添加作为外部JAR文件的二进制插件.外部jar可以使用 `buildscrip{}` 块添加到构建脚本类路径就像Section 62.6, “External dependencies for the build script”中描述的一样.

Example 21.4. Applying a plugin with the buildscript block

build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath "com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1"
    }
}

apply plugin: "com.jfrog.bintray"
```

使用插件的插件DSL

> 插件DSL正在孵化(incubating)中,请注意,在以后的Gradle版本中,DSL和其它配置可能会改变.

新的插件DSL提供了更为简洁,方便的方式来声明插件的依赖关系。它的适用于与新的Gradle Plugin Portal，同时提供了方便的核心和社区插件.该插件脚本块配置PluginDependenciesSpec的实例.

要应用的核心插件,可以使用短名称:

Example 21.5. Applying a core plugin

build.gradle

```
plugins {
    id 'java'
}
```

要从插件门户应用一个社区插件,必须使用插件的完全限定id:

Example 21.6. Applying a community plugin

build.gradle

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

不必要进行进一步的配置,就是说没有必要配置buildscript的类路径,Gradle会从插件门户找到该插件,并使构建可用.

参见PluginDependenciesSpec查看关于使用插件DSL的更多信息。

插件DSL的限制

想项目中添加插件的新方法不仅仅是一种更为方便的语法.新的DSL语法处理与老方法有很大的不同.新的机制允许Gradle更早更快的确定插件.这允许做更智能的事,如:

- 优化插件类的加载和重用.
- 允许不同的插件使用不同版本的依赖关系.
- 为编辑构建脚本提供关于隐含属性和值的详细信息

这要求插件被指定使Gradle在执行之前剩下的构建脚本前可以很容易并且很快的提取它.还要求插件使用一些静态的定义。

新的插件机制与"传统的" `apply()` 方法有一些关键区别.也有一些限制,其中一些限制是临时的,随着机制的成熟会逐渐没有,还有一些是方法固有的.

约束语法

新插件 `{}` 块不支持任意Groovy代码.被限制的原因是为幂等(每次产生相同的结果)和无副作用(为了Gradle随时执行的安全).

形式是:

```
plugins{
    id «plugin id»  version «plugin version»
}
```

`«plugin id»` 和 `«plugin version»` 必须是常量,字面量,字符串.其他语句都是不允许的;他们的存在会导致编译错误.

插件 `{}` 块也必须在构建脚本的顶部声明.它不能被嵌套在另一个结构(例如,if语句或for循环).

只能在构建脚本中使用

插件 `{}` 块目前只能在一个项目的构建脚本中使用.他不能在脚本插件, `settings.gradle` 和出书画脚本中使用.

*Gradle*的未来版本将删除此限制.

不能与subjects{},allprojects{}等结合使用

目前不能使用一次添加一个插件到多个项目中的模式,如使用 `subprojects{}` 等方式.目前没有机制可以应用一次插件到多个项目.目前,每个项目都必须在自己的构建脚本中的 `plugins{}` 块中声明应用的插件.

*Gradle*的未来版本将删除此限制.

如果新语法的限制让人望而却步,推荐使用buildscript{} block.

查找社区插件

Gradle有一个充满活力的,卡法人员做出不用贡献的插件社区.该Gradle插件门户提供搜索和探索社区插件的接口.

更多关于插件

本章旨在作为介绍插件和Gradle,以及他们扮演的角色.要了解更过关于插件内部的工作原理,参见Chapter 61, Writing Custom Plugins.

Standard Gradle plugins

许多包括在Gradle分布的插件。这些在下面列出。

语言插件

这些插件添加了可以被编译并在JVM中执行的各种语言的支持

Table 22.1. Language plugins

Plugin Id	自动应用	协同工作	描述
java	java-base	-	为项目添加java编译,测试及绑定能力,作为许多Gradle插件的基础.参见Chapter 7, Java Quickstart
groovy	java, groovy-base	-	为Groovy项目构建增加支持,参见Chapter 9, Groovy Quickstart
scala	java, scala-base	-	增加了对Scala项目构建的支持
antlr	java	-	增加了对使用ANTLR的生成解析器的支持.

孵化中的语言插件

这些插件增加对各种语言的支持:

Table 22.2. Language plugins

Plugin Id	自动应用	协同工作	描述
assembler	-	-	增加了原生的汇编语言能力的项目。
c	-	-	添加C源代码编译能力的项目.
cpp	-	-	增加C ++源代码编译能力的项目.
objective-c	-	-	添加的Objective-C ++源代码编译能力的项目.
windows-resources	-	-	增加了对包括Windows资源的本机二进制文件的支持.

集成插件

这些插件提供的各种运行时的技术的集成.

Table 22.3. Integration plugins

Plugin Id	自动应用	协同工作	描述
application	java, distribution	-	增加了对运行绑定Java项目作为命令行应用的任务.
ear	-	java	增加了对构建J2EE应用程序的支持.
jetty	war	-	在构建中嵌入Jetty web容器可以部署web应用.参见Chapter 10, Web Application Quickstart
maven	-	java, war	增加了对发布artifacts到Maven仓库的支持.
sogi	java-base	java	增加了对构建OSGi支持
war	java	-	增加了对组装Web应用程序WAR文件的支持.参见Chapter 10, Web Application Quickstart

孵化中的集成插件

这些插件提供的各种运行时的技术的集成.

Table 22.4. Incubating integration plugins

Plugin Id	自动应用	协同工作	描述
distribution	-	-	对构建增加对ZIP和TAR的支持
java-library-distribution	java, distribution	-	增加了对建筑ZIP和TAR的一个Java库的支持.
ivy-publish	-	java, war	这个插件提供了一个新的DSL支持发布artifacts ivy存储库，它改善了现有的DSL.
maven-publish	-	java, war	这个插件提供了一个新的DSL支持发布artifacts Maven仓库，它改善了现有的DSL。

软件开发插件

这些插件在您的软件开发过程中提供帮助.

Table 22.5. Software development plugins

Plugin Id	自动应用	协同工作	描述
announce	-	-	消息发布到自己喜欢的平台，如Twitter或Growl.
build-announcements	announce	-	发送本地通知关于有趣的事件在构建生命周期到你的桌面.
checkstyle	java-base	-	使用Checksytle对项目的Java源码执行质量检测,并生成报告.
codenarc	groovy-base	-	使用CodeNarc对项目的Groovy的源文件进行质量检测,并生成检测报告
eclipse	-	java,groovy, scala	生成Eclipse IDE的文件,从而能够以导入项目到Eclipse.参见Chapter 7, Java Quickstart
eclipse-wtp	-	ear, war	与eclipse插件一样,生成eclipse WPT(Web Tools Platform)配置文件, 导入到Eclipse中war/ear项目应配置与WTP工作.参见参见Chapter 7, Java Quickstart
findbugs	java-base	-	使用FindBugs执行项目的Java源文件质量检测,并生成检测报告
idea	-	java	生成Intellij IDEA IDE配置文件,从而可以将项目导入IDEA。
jdepend	java-base	-	使用JDepend执行项目的源文件质量检测,并生成检测报告
pmd	java-base	-	使用PMD执行项目的源文件质量检测,并生成检测报告
project-report	reporting-base	-	生成一个包含关于您的Gradle构建有用信息的报告。
signing	base	-	添加数字签名档案和artifacts的能力。
sonar	-	java-base, java, jacoco	与Sonar代码质量平台整合.由sonar-runner插件提供

孵化中的软件开发插件

这些插件在您的软件开发过程中提供帮助.

Table 22.6. Software development plugins

Plugin Id	自动应用	协同工作	描述
build-dashboard	reporting-base	-	生成构建仪表板报告.
build-init	wrapper	-	对Gradle初始化一个新构建提供支持.将一个Maven构建转换为Gradle构建
cnuit	-	-	提供运行CUnit测试支持
jacoco	reporting-base	java	对面向Java的JaCoCo代码库整合
sonar-runner	-	java-base, java, jacoco	提供与Sonar代码质量平台的整合.取代sonar插件
visual-studio	-	-	增加了与Visual Studio集成.
wrapper	-	-	增加一个Wrapper任务来生成Gradle打包文件.
java-gradle-plugin	java	-	通过提供标准的插件生成配置和验证,协助Gradle发展.

基础插件

这些插件是形成其他插件的基本构建模块.你可以在你的构建文件中使用它们,在下面李处完整地列表,然而,注意它们还不是Gradle的公用API的一部分.因此,这些插件未记录在用户指南中.你可能会参考他们的API文档,详细了解它们.

Table 22.7. Base plugins

base

添加标准的生命周期任务和配置合理的默认归档任务:

- 增加*ConfigurationName*任务.这些任务组装指定配置的工件。
- 增加了上传*ConfigurationName*任务,这些任务组装并上传指定配置的工件。
- 对所有归档任务配置合理的默认值(如继承AbstractArchiveTask的任务).如归档类型的任务:Jar,Tar,Zip.特别的,归档的 `destinationDir` , `baseName` 和 `version` 属性是预先配置的默认值.这是非常有用的,因为它推动了跨项目的一致性;关于档案的命名规则和完成构建后的位置的一致性。

java-base

- 增加资源集的概念到项目中.不添加特定的资源.

groovy-base

- 增加了Groovy的源集理念到项目中.

scala-base

- 添加scala源集合概念到项目中.

reporting-base

- 为项目增加了一些涉及到生产报告的公约性质的属性,

> 译者注:实在不会使用MarkDown在表格中加入列表,好在只表格只有两列,故本节不能按照官网的User Guide表格给出.而是以列表形式完成

第三方插件

你可以在Gradle Plugins site找到外部插件.

第 22 章 Java 插件

Java 插件给项目增加了编译, 测试以及打包的能力, Gradle 的许多其他插件都需要以 Java 插件为基础.

用法

要使用 Java 插件, 需要在构建脚本中加入如下内容

例子 **22.1.**使用 **Java** 插件

bulid.gradle

```
apply plugin: 'java'
```

资源设置

Java 插件引入了资源设置 (Source Set) 的概念, 资源设置就是一组被编译和执行在一起的源文件. 这些源文件可能包含 Java 的源文件以及一些资源文件. 其他的插件可能还会在资源设置中包含 Groovy 和 Scala 的源文件. 资源设置有一个与之关联的关于编译的 classpath 和有关运行的 classpath.

资源设置的用法之一就是将源文件归档到描述它们目的的各个逻辑组, 举个例子, 你可以使用一个资源设置来定义一个集成测试套件 也可以使用另外的资源设来定义你项目的 API 和实现类.

Java 插件定义了两个标准资源设置, 分别称为 `main` 和 `test` , `main` 资源设置中包含最终面向客户的代码, 也就是编译和集成为一个 Jar 文件. `test` 资源设置包括了测试阶段的代码, 也就是使用 JUnit 或者 TestNG 编译和执行的代码. 它们可以是单元测试, 集成测试, 验收测试或者任何对你有用的测试集.

任务

Java 插件引入了许多任务到项目当中, 具体如下表所示

表**22.1 java** 插件**-**任务

任务名	依赖	类型	描述
compileJava	所有产生编译 classpath 的任务，包括编译配置项目的所依赖的 jar 文件	JavaCompile	使用 javac 命令编译产生 java源文件
processResources	-	Copy	复制生产资源到生产 class 文件目录
classes	compileJava任务和processResources任务。有一些插件添加额外的编译任务	Task	组装生产class文件目录
compileTestJava	compile任务加上所有产生测试编译的classpath的任务	JavaCompile	使用 javac编译产生 java 测试源文件
processTestResources	-	Copy	复制测试资源到测试 class 文件目录
testClasses	compileTestJava 和processTestResources任务。一些插件会添加额外的测试编译任务	Task	组装测试class文件目录
jar	compile	Jar	组装 Jar 文件
javadoc	compile	javadoc	使用 javadoc 命令为 Java 源码生产 API 文档
test	compile，compileTest，加上所有产生 test runtime classp 的任务	Test	使用 JUnit或者 TestNG 进行单元测试
uploadArchives	在archives配置中产生信息单元的文件，包括了 jar	Upload	上传信息单元在archives配置中，包括 Jar 文件
clean	-	Delete	删除项目构建目录
clean*TaskName*	-	Delete	删除指定任务名所产生的项目构建目录，CleanJar会删除jar任务创建的 jar 文件，cleanTest将会删除由 test 任务创建的测试结果

对于添加到项目中的每个资源设置, java 插件将会加入以下编译任务

表**22.2.java** 插件**-**资源设置任务

任务名	依赖	类型	描述
compile*SourceSet*Java	产生资源设置编译 classpath 的所有任务	JavaCompile	使用 javac 命令编译给定资源设置的 Java 源文件
processSourceSetResources	-	Copy	复制给定资源设置的资源到 classes 目录下。
sourceSetClasses	compileSourceSetJava任务和 processSourceSetResources 任务。一些插件给资源设置添加额外的编译工作。	Task	组装资源设置的class目录

Java 插件同时也增加了一些为项目生命周期服务的任务

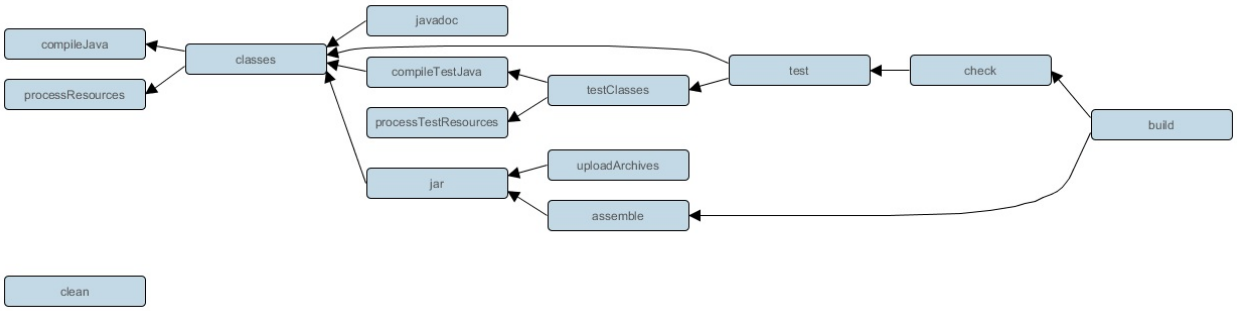
表**22.3.java** 插件**-**生命周期任务

任务名	依赖	类型	描述	
assemble	项目中的所有归档任务，包括 jar 任务。一些插件给项目增加的额外归档任务	Task	组装项目的所有档案	
check	项目中的所有验证任务，包括 test 任务。一些插件给项目增加的额外验证任务	Task	执行项目中的所有验证任务	
build	assemble任务和 check 任务		Task	构建完整地项目
buildNeeded	build 任务和buildNeeded 任务的testRuntime任务配置的所有项目的依赖库	Task	构建完整地项目并且构建该项目依赖的所有项目	
buildDependents	build and buildDependents tasks in all projects with a project lib dependency on this project in a testRuntime configuration.	Task	构建完整项目并且构建所有依赖该项目的项目	
buildConfigName	产生由ConfigName配置的信息单元的任务。	Task	根据指定的配置组装信息单元。这个任务是由 Java 插件隐式添加的基础插件添加的。	
uploadConfigName	上传由ConfigName配置的信息单元的任务。	Upload	根据指定的配置组装并上传信息单元。	

。这个任务是由 Java 插件隐式添加的基础插件添加的。

下图显示了这些任务之间的关系

图**22.1.java** 插件**-**任务


javadoc
compileJava
classes
compileTestJava
test
check
processResources
testClasses
processTestResources
build
uploadArchives
jar
assemble
clean

项目布局

Java 插件的默认布局如下图所示, 无论这些文件夹中有没有内容, Java 插件都会编译里面的内容, 并处理任何缺失的内容.

表**22.4.java** 插件**-**默认布局

目录 | 含义 ---— | --- src/main/java | 主要 Java 源码 src/main/resources | 主要资源 src/test/java | 测试 Java 源码 src/test/resources | 测试资源 src/sourceSet/java | 指定资源设置的 Java 源码 src/sourceSet/resources | 指定资源设置的资源

1. 改变项目布局

可以通过配置适当的资源设置来配置项目布局, 更多细节会在后面的章节中讨论, 下面是一个配置了 java 和 resource 的简单的例子

例**22.2.**自定义 **Java** 源码布局

build.gradle

```
sourceSet{
  main{
    java{
      srcDir 'src/java'
    }
    resources{
      srcDir 'src/resources'
    }
  }
}
```

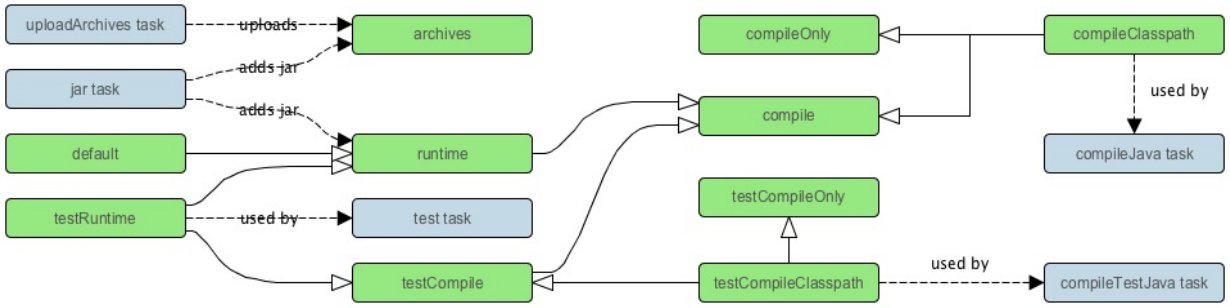
依赖管理

Java 插件给项目增加了许多关于依赖的配置, 如下所示, 这些配置被分配给许多任务, 比如 compileJava 和 test 等配置

表**22.5.Java**插件**-**依赖配置

名称	扩展	被使用时运行的任务	含义
compile	-	compileJava	编译时的依赖
runtime	compile	-	运行时的依赖
testCompile	compile	compileTestJava	编译测试所需的额外依赖
testRuntime	runtime	test	仅供运行测试的额外依赖
archives	-	uploadArchives	项目产生的信息单元（如:jar包）
default	runtime	-	使用其他项目的默认依赖项，包括该项目产生的信息单元以及依赖

图**22.2.Java**插件**-**依赖配置



对于每个添加到该项目的资源设置，java 插件会添加以下的依赖配置

表**22.6.Java**插件**-**资源设置依赖关系配置

名称	扩展	被使用时运行的任务	含义
sourceSetCompile	-	compileSourceSetJava	编译时给定资源设置的依赖
sourceSetRuntime	-	-	运行时给定资源设置的依赖

公共属性

Java 插件会为项目添加一系列的公共属性, 如下所示, 你可以在构建脚本中像项目属性那样直接使用它们 (see ???).

表**22.7.Java**插件**-**目录属性

属性名称	类型	默认值	描述
reportsDirName	String	reports	在构建目的生成报的文件夹
reportsDir	File (read-only)	buildDir/reportsDirName	该目录下生成报告
testResultsDirName	String	test-results	在构建目的测试结的result.x的存放目名
testResultsDir	File (read-only)	buildDir/testResultsDirName	测试结果result.xm文件会存在该文件中
testReportDirName	String	tests	在构建目的测试报的文件夹
testReportDir	File (read-only)	reportsDir/testReportDirName	测试的测报告会存在该目录
libsDirName	String	libs	在构建目下的类库件夹名
libsDir	File (read-only)	buildDir/libsDirName	该目录下放类库
distsDirName	String	distributions	在构建目下的distributio文件夹名
distsDir	File (read-	buildDir/distsDirName	该目录下放生成的

	only)		distributic
docsDirName	String	在构建目录下的doc文件夹名	
docsDir	File (read-only)	buildDir/docsDirName	该目录下放生成的档
dependencyCacheDirName	String	dependency-cache	在构建目下的依赖存文件夹
dependencyCacheDir	File (read-only)	buildDir/dependencyCacheDirName	该目录用缓存源依信息。

表**22.8.Java**插件**-**其他配置

属性名称	类型	默认值	描述
sourceSets	SourceSetContainer	Not null	包含项目的资源设置
sourceCompatibility	JavaVersion.也可以使用String类型或Number类型,如'1.5' 或 1.5	当前使用的JVM版本	编译Java源码时所使用的Java兼容版本
targetCompatibility	JavaVersion.也可以使用String类型或Number类型,如'1.5' 或 1.5	sourceCompatibility	生成class文件的Java版本
archivesBaseName	String	projectName	用于.jar文件或者.zip存档的基本名称
manifest	Mainfest	an empty manifest	该清单中包括所有的JAR文件

按照JavaPluginConvention和BasePluginConvention类型提供这些属性.

使用资源设置

你可以通过 sourceSets 属性来使用资源设置. 它其实是一个项目资源设置的容器, 它的类型是 SourceSetContainer. 同时也有 sourceSets { } 脚本模块, 在这个模块里, 你可以通过闭包配置资源设置容器. 资源设置容器的工作方式和其余容器几乎一模一样, 比如 tasks.

例 **22.3.** 进入资源设置

build.gradle

```
// Various ways to access the main source set
println sourceSets.main.output.classesDir
println sourceSets['main'].output.classesDir
sourceSets {
    println main.output.classesDir
}
sourceSets {
    main {
        println output.classesDir
    }
}

// Iterate over the source sets
sourceSets.all {
    println name
}
```

为了配置一个已经存在的资源集, 你只需要使用上面提及的访问方法来设置资源集的属性. 下面就是一个配置 Java 资源目录的例子:

例 **22.4.** 配置资源集的源目录

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

22.7.1.Source Set 属性

下表列出了 Source Set 的一些重要属性, 更多细节请查看 SourceSet 的 API 文档.

表22.9.java 插件- Source Set 属性

配置名称	类型	默认值	
name	String (read-only)	Not null	用 so se
output	SourceSetOutput(read-only)	Not null	so se 文 其 cla re
output.classesDir	File	buildDir/classes/name	在 下 放 so se cla 件
output.resourcesDir	File	buildDir/resources/name	在 下 放 so se re 文
compileClasspath	FileCollection	compileSourceSet configuration	这 so se 使 cla
runtimeClasspath	FileCollection	output + runtimeSourceSet configuration	执 so se cla 件 cla
	SourceDirectorySet(read-		当 so se 源 包

java	SourceDirectorySet(read-only)	Not null	于录有件他件
java.srcDirs	Set.可以设置为在Section 15.5, “Specifying a set of input files”中描述的任何值	[projectDir/src/name/java]	该se jav件
resources	SourceDirectorySet(read-only)	Not null	该se源存re目资会re下有件件件Gr件集除他
resources.srcDirs	Set.可以设置为在Section 15.5, “Specifying a set of input files”中描述的任何值	[projectDir/src/name/resources]	该se资的
allJava	SourceDirectorySet(read-only)	java	该se有件插Gr件额Ja件集
			该se源这有文

allSource	SourceDirectorySet(read-only)	resources + java	文些如插加源这合

22.7.2.定义一个新的 source set

要定义一个新的源组, sourceSets {} 块中引用它.下面是一个例子：

例22.5.定义一个新的 source set

build.gradle

```
sourceSets {
    intTest
}
```

当你定义一个新的 source set, java 插件会为该 source set 添加一些如Table 22.6, "Java plugin - source set dependency configurations"中所示的依赖配置关系.可以使用这些配置来定义source set的编译和运行时依赖。

例22.6.定义 source set 的依赖

build.gradle

```
sourceSets {
    intTest
}

dependencies {
    intTestCompile 'junit:junit:4.12'
    intTestRuntime 'org.ow2.asm:asm-all:4.0'
}
```

java 插件增加了一些如Table 22.2, "Java plugin - source set tasks"为该source set组装classes文件的任务,例如，对于一个叫intTest的source set，为此source set编译classes任务运行 `gradle intTestClasses` 完成。

例22.7.编译一个 source set

`gradle intTestClasses` 命令的输出

```
> gradle intTestClasses
:compileIntTestJava
:processIntTestResources
:intTestClasses

BUILD SUCCESSFUL

Total time: 1 secs
```

22.7.3.一些 source set 的例子

加入含有类文件的 sorce set 的 JAR：

例22.8.为 source set 组装 JAR

build.gradle

```
task intTestJar(type: Jar) {
    from sourceSets.intTest.output
}
```

为 source set 生成 javadoc:

例22.9.为 source set 生成 javadoc

build.gradle

```
task intTestJavadoc(type: Javadoc) {
    source sourceSets.intTest.allJava
}
```

为 source set 添加一个测试套件运行测试：

例22.8.source set 运行测试

build.gradle

```
task intTest(type: Test) {
    testClassesDir = sourceSets.intTest.output.classesDir
    classpath = sourceSets.intTest.runtimeClasspath
}
```

22.8.Javadoc

Javadoc task 是 Javadoc 的一个实例. 它支持 Javadoc 的核心选项和可执行的 Javadoc 的 reference documentation 中描述的标准 J avaTOC 的选项. 有关支持 Javadoc 选项的完整列表, 请参阅以下类的API文档：CoreJavadocOptions 和 StandardJavadocDocletOptions.

表**22.10.java** 插件**- javadoc** 配置

任务属性	类型	默认值
classpath	FileCollection	sourceSets.main.output + sourceSets.main.compileClasspath
source	FileTree.可以设置为在Section 15.5, “Specifying a set of input files”中描述的任何值	sourceSets.main.allJava
destinationDir	File	docsDir/javadoc
title	String	project 的 name 和 version

22.9.Clean

clean 任务是一个 Delete 的实例. 它只是删除 dir 属性指定的目录.

表**22.11.java** 插件 **- Clean** 的属性

任务属性	类型	默认值
dir	File	buildDir

22.10. 资源

Java 插件使用 Copy 任务处理资源. 它为项目每个 source set 都增加了一个实例. 可以参考Section 15.6, "Copying files" 获取关于copy任务的信息.

表**22.12.java** 插件**- ProcessResources** 的属性

任务属性	类型	默认值
srcDirs	Object.可以在Section 15.5, “Specifying a set of input files”中查看使用什么	sourceSet.resources
destinationDir	File.可以再Section 15.1, “Locating files”查看使用什么	sourceSet.output.resourcesDir

22.11.编译 java

java 插件为项目的每一个 source set 增加了一个 JavaCompile 实例, 最常见的配置选项如下所示:

表**22.13.java** 插件**-**编译配置

任务属性	类型	默认值
classpath	FileCollection	sourceSet.compileClasspath
source	FileTree,可以在Section 15.6, “Copying files”中查看可以设置什么.	sourceSet.java
destinationDir	File.	sourceSet.output.classesDir

默认情况下 java 的编译运行在 Gradle 中的进程. 设置 option.fork 为 true 会使编译在一个单独的进程中运行,在Ant中运行 javac任务意味着一个新进程将被拆封为多个编译任务，这会减慢编译。相反的,Gradle的直接编译集成（见上文）在编译过程中将尽可能地重复使用相同的进程.在所有情况下由options.forkOptions指定的选项会被实现.

22.12.增量Java编译

从Gradle2.1开始,可以使用Java增量编译,此功能正在孵化,参见JavaCompile如何启用这个功能. 增量编译的主要目标如下:

- 避免在没必要编译的java编译资源上浪费时间.这意味着更快构建,尤其是在改变一些class与jar的时候,不需要再次编译那些不依赖这些class与jar的文件.
- 尽可能地少输出class.类不需要重新编译意味着保持输出目录不变。一个示例场景中，真正使用JRebel的真正有用的是 - 越少的输出类被改变，JVM可以使用越快刷新。

更高级的增量编译:

- 检测陈旧类的设置是否正确是以牺牲速度为代价的,该算法分析字节码并与编译器直接交互(非私有常量内联),依赖传递等.举个例子:当一个类的公共常量改变后,我们希望避免由编译器编译内联常数产生的问题,我们将调整算法和缓存以便增量Java编译可以是每编译任务的默认设置。
- 为了使增量编译快，我们缓存会分析class的结果和jar快照。最初的增量编译应该会慢于cold caches.

22.13.测试

test任务是Test的一个实例.它会在测试source set自动检测并执行所有的单元测试,并且在测试执行完成后会生成一份测试报告.task任务支持JUnit和TestNG.在Test查看完整地API.

22.13.1.执行测试

测试从main构建过程中分离出来的,运行在一个单独的JVM中执行.Test任务允许控制这些如何发生. 有许多属性用于控制测试过程如何启动.这包括使用诸如系统属性，JVM参数和Java可执行文件。

可以指定是否要并行执行测试.Gradle通过同时运行多个测试进程提供并行执行测试.每个测试进程在同一时间只能执行一个测试,为了充分利用这一特性,一般不需要为tests任务做什么特别的设置, `maxParallelForks` 属性指定测试进程在同一时间运行的最大进程数.默认值是1,意味着不执行并行测试.

测试过程中设置org.gradle.test.worker系统属性为该测试过程的唯一标识符,例如，在文件名或其他资源标识符的唯一标识符。

你可以指定一些测试任务在已执行了一定数量的测试后重新运行.这可能是一个非常好的方式替代测试进程中的大量的堆.forkEvery属性指定测试类的在测试过程执行的最大数目。默认的是执行在各测设进程中不限数量的测试。

该任务有一个ignoreFailures属性来控制在测试失败时的行为。测试任务总是执行每一个检测试验.它停止构建之后，如果ignoreFailures是false，说明有失败的测试。ignoreFailures的默认值是false。

testLogging属性允许你配置哪个测试事件将被记录，并设置其log等级。默认情况下，将记录每一个失败的测试简明消息。详见TestLoggingContainer如何按需求调整测试记录。

22.13.2.调试

测试任务提供了Test.getDebug()属性，可使JVM等待调试器附加到5005端口后在进行调试.

通过调用 `--debug-JVM` 任务选项，这也可以启用调试任务（since Gradle1.12）。

22.13.3.测试过滤

从Gradle1.10开始,可以根据测试任务名进行特点的任务测试,过滤与在构建脚本的段落中引入/排除测试任务(-Dtest.single, test.include and friends)是两种不同的机制.后者是基于文件,如测试实现类的物理位置.选择文件级的测试会不支持那些被测试等级过滤掉的一些有趣的测试脚本.下面的这些有些已经被实现,有些是将来会实现的:

- 过滤特定等级的试验方法;执行单个测试方法
- 通配符"*"支持任意字符匹配
- 命令行选项"--tests"用以设置测试过滤器.对经典的'执行单一测试方法'用例尤其有用.当使用命令行选项的时候,在构建脚本中声明的包含过滤器被忽略。
- Gradle过滤测试框架API的限制.一些高级的综合性测试无法完全兼容测试过滤,但是,绝大多是测试和用例可以被熟练地处理.
- 测试过滤取代基于选择文件的测试,后者在未来可能会完全被取代.我们会完善测试过滤的API,并增加不同的过滤器

例**22.11.**过滤测试的构建脚本吗 **build.gradle**

```
test{
    filter{
        //包括在测试的任何具体方法
        includeTestsMatching "*UiCheck"

        //包括所有包种的测试
        includeTestsMatching "org.gradle.internal.*"

        //包括所有的集成测试
        includeTestsMatching "*IntegTest"
    }
}
```

要了解更多详细信息和示例，请参见TestFilter。 使用命令行选项的一些例子:

- gradle test --tests org.gradle.SomeTest.someSpecificFeature
- gradle test --tests *SomeTest.someSpecificFeature
- gradle test --tests *SomeSpecificTest
- gradle test --tests all.in.specific.package*
- gradle test --tests *IntegTest
- gradle test --tests *IntegTest*ui*
- gradle someTestTask --tests *UiTest someOtherTestTask --tests *WebTest*ui

22.13.4.通过系统属性执行单独测试

> 如上所述该机制已经被测试过滤取代
>
> 译者注:被取代的东西就先不翻译了

Setting a system property of taskName.single = testNamePattern will only execute tests that match the specified testNamePattern. The taskName can be a full multi-project path like ":sub1:sub2:test" or just the task name. The testNamePattern will be used to form an include pattern of "**/testNamePattern*.class";. If no tests with this pattern can be found an exception is thrown. This is to shield you from false security. If tests of more than one subproject are executed, the pattern is applied to each subproject. An exception is thrown if no tests can be found for a particular subproject. In such a case you can use the path notation of the pattern, so that the pattern is applied only to the test task of a specific subproject. Alternatively you can specify the fully qualified task name to be executed. You can also specify multiple patterns. Examples:

- gradle -Dtest.single=ThisUniquelyNamedTest test
- gradle -Dtest.single=a/b/ test
- gradle -DintegTest.single=*IntegrationTest integTest
- gradle -Dtest.single=:proj1:test:Customer build
- gradle -DintegTest.single=c/d/ :proj1:integTest

22.13.5.测试检测

测试任务检测哪些类是通过检查编译测试类的测试类。默认情况下它会扫描所有.calss文件.可以自定义包含/排除哪些类需不要要被扫描.所使用不同的测试框架（JUnit/ TestNG）时测试类检测使用不同的标准。 当使用JUnit，我们扫描的JUnit3和JUnit4的测试类。如果任一下列条件匹配，类被认为是一个JUnit测试类：

- 类或父类集成自TestCase或GroovyTestCase
- 类或父类有@RunWith注解
- 类或者父类中的方法有@Test注解

当使用TestNG的，我们扫描注解了@Test的方法。

需要注意的是抽象类不执行。Gradle还扫描了继承树插入测试classpath中的jar文件。

如果你不想使用测试类的检测，可以通过设置scanForTestClasses为false禁用它。这将使得测试任务只使用包含/排除找到测试类。如果scanForTestClasses是false而且额并没有包含/排除指定模式,"**/*Tests.class","**/*Test.class",与"**/*Abstract*.class"分别为包含/排除的默认值.

22.13.6.测试分组

JUnit和TestNG允许为测试方法精密分组.

对于分组JUnit的测试类与测试方法,JUnit4.8引入了类别的概念.[9]该测试任务允许您设定JUnit包括或者排除某些类的规范。

例**22.12.JUnit**分类 **build.gradle**

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

TestNG的框架有一个非常类似的概念。在TestNG的，你可以指定不同的测试组。[10]测试分组应包括或排除在测试任务进行配置了的测试执行.

例**22.13.TestNG**分组测试 **build.gradle**

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

22.13.7.测试报告

测试任务默认生成以下结果.

- 一份HTML测试报告
- 一个与Ant的JUnit测试报告任务兼容的XML.这个格式与许多其他服务兼容,如CI serves
- 结果是有效的二进制,测试任务会从这些二进制结果生成其他结果。 有一个独立的TestReport任务类型会根据一些Test任务实例生成的二进制源码生成一个HTML报告.使用这种测试类型,需要定义一个destinationDir,里面包括测试结果的报告.下面是一个示例，它产生一个从子项目的单元测试组合而成的报告： 例**22.14.**创建单元测试报告子项目 **build.gradle** ``` subprojects { apply plugin: 'java'

  // Disable the test report for the individual test task test {

  ```
  reports.html.enabled = false
  ```

  } }

task testReport(type: TestReport) { destinationDir = file("$buildDir/reports/allTests") // Include the results from the `test` task in all subprojects reportOn subprojects*.test } ``` 应该注意的是，TestReport型组合来自多个测试任务的结果，需要聚集个别测试类的结果。这意味着，如果一个给定的测试类是由多个测试任务执行时，测试报告将会包括那些类,但是很难区分该输出结果分别是出自哪个类.

22.13.7.1.TestNG 的参数化方法和报告

TestNG支持参数化方法,允许一个特定的测试方法使用不同的输入被执行多次。Gradle会在测试报告中包含该方法的参数值. 给出一个叫aTestMethod的测试方法,该方法有两个参数,在测试报告中会根据名字报告:aTestMethod(toStringValueOfParam1, toStringValueOfParam2). 这很容易识别的参数值的特定迭代.

22.13.8.公共值

表22.14.Java插件-测试属性

任务属性	类型	默认值
testClassesDir	File	sourceSets.test.output.classesDir
classpath	FileCollection	sourceSets.test.runtimeClasspath
testResultsDir	File	testResultsDir
testReportDir	File	testReportDir
testSrcDirs	List	sourceSets.test.java.srcDirs

22.14.Jar

jar 任务创建包含项目的类文件和资源的 JAR 文件. JAR 文件在 archives 的依赖配置中是作为一个 artifact 的声明. 这意味着, JAR 是相关项目一个可用的 classpath. 如果您上传您的项目到存储库, 这个 JAR 会被声明为依赖描述符的一部分. 可以再Section 15.8, “Creating archives”与Chapter 51, Publishing artifacts中了解更多关于 JAR 与 archives 与 artifact configurations 协同工作的更多细节.

22.14.1.Manifest

每个 jar 或 war 对象有一个 manifest 属性做为Manifest单独的实例, 当生成存档, 一个对应 MANIFEST.MF文件被写入到档案中.

例**22.15.MANIFEST.MF**的定制
build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
                   "Implementation-Version": version)
    }
}
```

你可以创建一个 manifest 的独立实例. 您可以使用如共享 jar 之间的 manifest 的信息.

例**22.16.**创建一个**manifest**对象
build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle",
               "Implementation-Version": version)
}
task fooJar(type: Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

您可以合并其他 manifest 到任何 Manifest 对象. 其它清单可能是通过文件路径描述或着像上所述, 引用另一个Manifest对象.

例**22.17.**独立的**MANIFEST.MF**一个特定的归档
build.gradle

```
task barJar(type: Jar) {
    manifest {
        attributes key1: 'value1'
        from sharedManifest, 'src/config/basemanifest.txt'
        from('src/config/javabasemanifest.txt',
             'src/config/libbasemanifest.txt') {
            eachEntry { details ->
                if (details.baseValue != details.mergeValue) {
                    details.value = baseValue
                }
                if (details.key == 'foo') {
                    details.exclude()
                }
            }
        }
    }
}
```

清单合并的顺序与声明语句的顺序相同,如果基本清单和合并的清单都为相同的密钥定义值，那么那么合并清单将会被合并,您可以通过添加在其中您可以使用一个ManifestMergeDetails实例为每个条目实体完全自定义的合并行为。声明不会立即被来自触发合并。这是延迟执行的，要么产生jar时，或要求写入effectiveManifest时. 你可以很容易地写一个清单到磁盘。 例 **22.17.**独立的**MANIFEST.MF**一个特定的存档 **build.gradle**

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

22.15.上传

如何上传archives在Chapter 51, Publishing artifacts

[9]JUnit的维基包含有关如何使用JUnit类工作的详细说明:https://github.com/junit-team/junit/wiki/Categories.

[10]TestNG的文档包含关于测试组的更多详细信息:http://testng.org/doc/documentation-main.html#test-groups.

War 插件 (未完成)

WAR插件扩展了Java插件,支持web应用组装成War文件.它默认禁用了Java插件JAR归档任务，并增加了一个默认的WAR归档任务。

25.1.使用

使用war插件需要在构建脚本下包括以下内容

例**25.1.**使用**war**插件

build.gradle

```
apply plugin 'war'
```

25.2.任务

War插件会添加下列任务到项目.

表**25.1.War**插件-任务

任务名	依赖	类型	描述
war	compile	War	组装应用程序War文件

War插件由Java插件添加下列依赖任务.

表**25.2.War**插件-附加的依赖任务

任务名	依赖
assemble	war

图**25.1.War**插件-任务

25.3.项目布局

表**25.3.War**插件-项目布局 文件夹 | 含义 -------- | ------ src/main/webapp | Web应用资源

25.4.依赖管理

War插件增加了名为providedCompile和providedRuntime的两个依赖配置.这两个配置有相同的作用域在编译或者运行时的配置,不同之处在于是否会将war文件归档.很重要的一点是它们都会提供配置传递.比如在任意的provided配置中添加了 `commons-httpclient:commons-httpclient:3.0` ,该依赖依赖于 `commons-codec` ,因为这个一个"provided"的配置,意味着这两个依赖都不会被加入你的WAR中,即使 `commons-codec` 库是一个显式的编译配置.如果不希望出现这种传递行为, `commons-httpclient:commons-httpclient:3.0@jar` 这样声明provided依赖即可.

25.5.公共配置

表**25.4.War**插件**-**目录配置

属性名称	类型	默认值	描述
webAppDirName	String	src/main/webapp	在项目目录的web应用的资源文件夹名
webAppDir	File (read-only)	projectDir/webAppDirName	Web应用的资源路径

这些属性由一个WarPluginConvention公共对象提供

25.6.War

War任务默认会把 `src/main/webapp` 的内容复制到归档目录的根目录.webapp文件夹下会包含一个 `WEB-INF` 子文件夹,里面可能会有一个web.xml文件.编译后的class文件会在 `WEB-INF/classes` 下,所有runtime[13]的依赖配置会被拷贝至 `WEB-INF/lib` 下.

API文档中有更多关于War的信息.

25.7.定制War

下面的例子中有一些重要的自定义选项

例**25.2.**定制**War**插件

build.gradle

```
configuration{
  moreLibs
}

respositories{
  faltDir {dirs "lib"}
  mavenCentral()
}

dependencies{
  compile module(":compile:1.0") {
        dependency ":compile-transitive-1.0@jar"
        dependency ":providedCompile-transitive:1.0@jar"
  }
  providedCompile "javax.servlet:servlet-api:2.5"
    providedCompile module(":providedCompile:1.0") {
        dependency ":providedCompile-transitive:1.0@jar"
  }
  runtime ":runtime:1.0"
  providedRuntime ":providedRuntime:1.0@jar"
  testCompile "junit:junit:4.12"
  moreLibs ":otherLib:1.0"
}

war{
  from 'src/rootContent' // 增加一个目录到归档根目录
  webInf {from 'src/additionalWebInf'} // 增加一个目录到 WEB-INF 下
  classpath fileTree('additionalLibs') // 增加一个目录到 WEB-INF/lib下.
  classpath configurations.moreLibs // 增加更多地设置到 WEB-INF/lib 下.
  webXml = file('src/someWeb.xml') // 复制xml文件到 WEB-INF/web.xml.
}
```

当然,可以用一个封闭的标签定义一个文件是否存打包到War文件中.

---

[13] `runtime` 配置将会继承 `compile` 配置.

使用

<<<<<<< HEAD

使用war插件需要在构建脚本下包括以下内容

例**25.1.**使用**war**插件

build.gradle

```
apply plugin 'war'
```

a655fa4be4421004591827ae70fee579703794dd

任务

<<<<<<< HEAD

War插件会添加下列任务到项目.

表**25.1.War**插件**-**任务

任务名	依赖	类型	描述
war	compile	War	组装应用程序War文件

War插件由Java插件添加下列依赖任务.

表**25.2.War**插件**-**附加的依赖任务

任务名	依赖
assemble	war

图**25.1.War**插件**-**任务

a655fa4be4421004591827ae70fee579703794dd

项目布局

表**25.3.War**插件**-**项目布局

文件夹	含义
src/main/webapp	Web应用资源

依赖管理

War插件增加了名为providedCompile和providedRuntime的两个依赖配置.这两个配置有相同的作用域在编译或者运行时的配置,不同之处在于是否会将war文件归档.很重要的一点是它们都会提供配置传递.比如在任意的provided配置中添加了 `commons-httpclient:commons-httpclient:3.0` ,该依赖依赖于 `commons-codec` ,因为这个一个"provided"的配置,意味着这两个依赖都不会被加入你的WAR中,即使 `commons-codec` 库是一个显式的编译配置.如果不希望出现这种传递行为, `commons-httpclient:commons-httpclient:3.0@jar` 这样声明provided依赖即可.

公共配置

表**25.4.War**插件-目录配置

属性名称	类型	默认值	描述
webAppDirName	String	src/main/webapp	在项目目录的web应用的资源文件夹名
webAppDir	File (read-only)	projectDir/webAppDirName	Web应用的资源路径

这些属性由一个WarPluginConvention公共对象提供

War

War任务默认会把 `src/main/webapp` 的内容复制到归档目录的根目录.webapp文件夹下会包含一个 `WEB-INF` 子文件夹,里面可能会有一个web.xml文件.编译后的class文件会在 `WEB-INF/classes` 下,所有runtime[13]的依赖配置会被拷贝至 `WEB-INF/lib` 下.

API文档中有更多关于War的信息.

---

[13] `runtime` 配置扩展了 `compile` 配置.

定制War

下面的例子中有一些重要的自定义选项

例25.2.定制War插件

build.gradle

```
configuration{
  moreLibs
}

respositories{
  faltDir {dirs "lib"}
  mavenCentral()
}

dependencies{
  compile module(":compile:1.0") {
        dependency ":compile-transitive-1.0@jar"
        dependency ":providedCompile-transitive:1.0@jar"
  }
  providedCompile "javax.servlet:servlet-api:2.5"
    providedCompile module(":providedCompile:1.0") {
        dependency ":providedCompile-transitive:1.0@jar"
  }
  runtime ":runtime:1.0"
  providedRuntime ":providedRuntime:1.0@jar"
  testCompile "junit:junit:4.12"
  moreLibs ":otherLib:1.0"
}

war{
  from 'src/rootContent' // 增加一个目录到归档根目录
  webInf {from 'src/additionalWebInf'} // 增加一个目录到 WEB-INF 下
  classpath fileTree('additionalLibs') // 增加一个目录到 WEB-INF/lib下.
  classpath configurations.moreLibs // 增加更多地设置到 WEB-INF/lib 下.
  webXml = file('src/someWeb.xml') // 复制xml文件到 WEB-INF/web.xml.
}
```

当然,可以用一个封闭的标签定义一个文件是否存打包到War文件中.

Ear 插件 (未完成)

Ear插件添加了对组装Web应用程序EAR文件的支持。 它将添加默认EAR文件生成任务。 它不需要Java插件，对于使用Java插件的项目，它会禁用默认的JAR文件生成。

51.1.使用

使用ear插件需要在构建脚本下包括以下内容

例**51.1.**使用**ear**插件

build.gradle

```
apply plugin 'ear'
```

51.2.任务

War插件会添加下列任务到项目.

表**51.2.War**插件**-**任务

任务名	依赖于	类型	描述
ear	compile (仅在java插件存在时)	Ear	组装应用程序Ear文件

Ear插件由其他已存在的插件添加下列依赖任务.

表**51.2.Ear**插件**-**附加的依赖任务

任务名	依赖于
assemble	ear

51.3.项目布局

表**25.3.Ear**插件**-**项目布局

文件夹	含义
src/main/application	Ear资源,比如**META-INF**目录

51.4.依赖管理

Ear插件添加了两个依赖配置：**deploy**和**earlib**。 **deploy**中的所有依赖关系都放在EAR存档的根目录中，并且是不可传递的（not transitive）。 **earlib**配置中的所有依赖关系都放在EAR存档中的'lib'目录中，并且是可传递的（transitive）。

51.5.常用配置

表**51.4.Ear**插件**-**目录配置

属性名称	类型	
appDirName	String	src/m
libDirName	String	lib
deploymentDescriptor	org.gradle.plugins.ear.descriptor.DeploymentDescriptor	在一个 署配置 applic

这些属性由一个EarPluginConvention公共对象提供

51.6.Ear

War任务默认会把`src/main/application`的内容复制到归档目录的根目录。如果配置文件**META-INF/application.xml**不存在，它将被自动生成。

API文档中有更多关于Ear的信息.

51.7.定制Ear

下面的例子中有一些重要的自定义选项

例**25.2.**定制**Ear**插件

build.gradle

```
pply plugin: 'ear'
apply plugin: 'java'

repositories { mavenCentral() }

dependencies {
    // The following dependencies will be the ear modules and
    // will be placed in the ear root
    deploy project(path: ':war', configuration: 'archives')

    // The following dependencies will become ear libs and will
    // be placed in a dir configured via the libDirName property
    earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'
}

ear {
    appDirName 'src/main/app'  // use application metadata found in this folder
    // put dependent libraries into APP-INF/lib inside the generated EAR
    libDirName 'APP-INF/lib'
    deploymentDescriptor {  // custom entries for application.xml:
//      fileName = "application.xml"  // same as the default value
//      version = "6"  // same as the default value
        applicationName = "customear"
        initializeInOrder = true
        displayName = "Custom Ear"  // defaults to project.name
        // defaults to project.description if not set
        description = "My customized EAR for the Gradle documentation"
//      libraryDirectory = "APP-INF/lib"  // not needed, above libDirName setting does
 this
//      module("my.jar", "java")  // won't deploy as my.jar isn't deploy dependency
//      webModule("my.war", "/")  // won't deploy as my.war isn't deploy dependency
        securityRole "admin"
        securityRole "superadmin"
        withXml { provider -> // add a custom node to the XML
            provider.asNode().appendNode("data-source", "my/data/source")
        }
    }
}
```

你也可以使用一些**Ear**任务提供的自定义选项，如**from**和**metaInf**.

51.8.使用自定义的配置文件

-/-/-/-/-//-/-/-/--/-/-/-/-/-当前进度↑↑**-/-/-/-/-/--/-/-/-/-/-/-/-/-/**